

START PROGRAMMING WITH THE ELECTRON

Masoud Yazdani



START PROGRAMMING WITH THE ELECTRON

MASOUD YAZDANI

Department of Computer Science, University of Exeter

Contributing authors

Henry Brinkworth

Michael Coker

Grenville Heath



Addison-Wesley Publishing Company

London Reading, Massachusetts Menlo Park, California Amsterdam
Don Mills, Ontario Manila Sydney Singapore Tokyo

Acknowledgements

I would like to thank John Coll, John Horton and Geoff Vincent, of Acorn Computers, for their continuous support as well as for the loan of a prototype Electron on which the programs were tested.

Any good ideas in this book which are not due to Henry Brinkworth, Michael Coker or Grenville Heath have been borrowed from the late Max Clowes, from Stephen Goudge, Jim Hunter, Ian MacCallum, Simon, Aaron Sloman, David Vines, or from the many people who acted as 'guinea pigs' during our experiments with earlier drafts. The job of transferring the ideas to text has depended on the super-efficient secretarial support of Marlene Teague. Thanks, too, to Simon for turning the whole thing into readable English. Only the errors are mine, and of them I claim sole ownership.

Masoud Yazdani

March 1983

Cartoons David Farris

Diagrams David Eaton

Cover design Stuart Hughes

© 1983 Addison-Wesley Publishers Limited

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise without prior written permission from the publisher.

ISBN: 201 14604 5

Phototypeset by Paston Press, Norwich

Printed in Finland

by Werner Söderström Osakeyhtiö

Member of Finnprint

FGHIJK 8987654

Preface

There are many ways of learning how to use a computer. One is by asking someone who has already got the ‘bug’, and it is this philosophy which underlies our intentions here. We hope that our book will provide you with a ‘hand-holding’ introduction to programming and to some of the capabilities of your new toy.

Another way of learning is by reading books that tell you how to use the different features of the computer. Such books, of which the *User Guide* is an example, usually tell you a great deal—often much more than you need to know in the beginning. However, once you have read this book and have a grasp of the basics, you should be able to find your way around this mass of useful information and discover how the system works by carrying out experiments.

We do recommend that, while working through this book, you refer to the relevant sections of the *User Guide*, in order to pick up more details about what you have been introduced to in each chapter.

Have fun.

Contents

1	Introducing the Acorn Electron	1
2	Writing a Song	17
3	Talking to Turtle	24
4	Playing with Numbers	31
5	Structured Problem Solving	39
6	Helping Turtle in a Maze	45
7	Input and Output	59
8	Collections of Objects	66
9	Have a Chat with your Micro	73
10	Sounding out your Electron	84
11	Pretty Pictures	93
12	Computer Games	109

Program listings

<i>Appendix A</i>	Turtle Graphics	118
<i>Appendix B</i>	Maze Solver	126
<i>Appendix C</i>	Greeter Program	132
<i>Appendix D</i>	Rivergame Program	134
Index		139



Introducing the Acorn Electron



Aims of this chapter

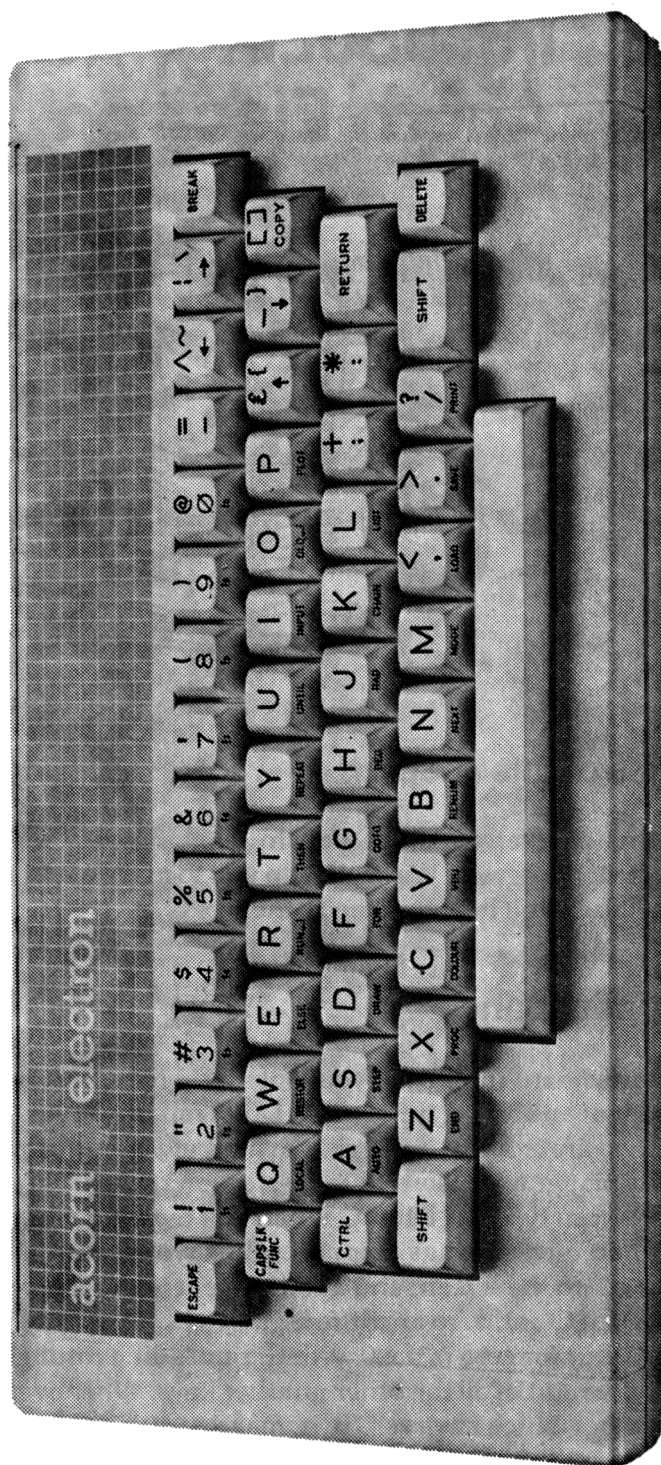
In this chapter we want you to learn how to write your very first **computer program**. We shall help you to make a few mistakes(!) and we shall teach you how to remedy them.

Introduction

The **keyboard** in front of you looks as shown in Figure 1.1. It is similar to that of an electric typewriter. Don't fall into the trap of believing that the two are exactly the same. On a typewriter, in some cases it doesn't really matter which key you press, so long as what is printed looks right. So you can type the letter O when you mean the number zero. But on a computer two things happen when you press a key. The symbol appears on the screen; and a code is sent to the computer itself. And the codes for O and zero are different.

The key for zero is the 0 with the cross-line through it (Ø). You'll find it to the right of all the other number keys. Whenever you mean to type a number which has a zero in it, make sure that you use this one—otherwise you'll get some very funny results! In the same way, make sure that

Fig. 1.1 The keyboard layout of the Acorn Electron.



you use the 1, rather than L or I when your number has a 1 in it. Finally, the long key at the bottom of the keyboard is the space-bar. Pressing this key gives you spaces.

By following the instructions in the *User Guide*, connect the computer to a TV set and then turn the computer on and the screen will display the following:

Acorn Electron ●

BASIC

>

If this message does not appear, your Electron is broken and you should ask somebody to fix it for you before you go any further.

The symbol > shows that the computer is asking you to type something in. In computer jargon the symbol > is called a **prompt**. Check that the (CAPS LK) light, at the left of the keyboard, is on. This light indicates that everything you type will appear in capital letters. If you wanted to type lower-case letters, such as a, b, c, rather than capitals, you could turn the light out by holding down the (SHIFT) key and pressing the (CAPS LK) key once. Press it again, and the light comes on again. For the time being, leave the (CAPS LK) on.

When you tap a key the character will appear on the screen in front of you. So now type in PLEASE SAY HELLO and then press the (RETURN) key. The computer responds with:

>PLEASE SAY HELLO

Mistake

>

The computer does not understand PLEASE SAY HELLO. Computers do not communicate in English as we do, but in their own languages, of which BASIC is one. PLEASE is not a command to the computer in the language BASIC, which we are using. The computer therefore has assumed that you have made a mistake.

You will soon be able to start to communicate with the computer in BASIC, but in the meantime carry on typing in messages, and practise using the keys and the (RETURN) key. If, after using the (RETURN) key, you do not get the > symbol on the screen, do not worry; just press the (BREAK) key. The meanings of these special keys, as well as the meaning of BASIC words, are explained briefly at the end of the chapter in which they are introduced and in detail in the *User Guide*.

Now type in the following:

>PRINT "HELLO" (To type " press the **SHIFT** key down and at the same time press the key with the **"2** on it.)

PRINT is a command which is part of the BASIC language. Because PRINT is a special BASIC **keyword** you need not type it one letter after another. If you want to, you can get PRINT by pressing the **FUNC** key and the **(?/)** key simultaneously. Most of the BASIC keywords can be typed completely with simultaneous strokes of the **FUNC** key and another key. On the front of each key you can read what that key would type if pressed together with the **FUNC** key.

The instruction will appear on the screen as you type it, but it will not be obeyed until you press the **RETURN** key. The computer will then respond:

```
HELLO
>
```

You have now started to communicate with the computer in BASIC. Suppose you wanted the computer to print another message; say

```
I AM A GOOD COMPUTER
```

but you made a mistake. Type:

```
>PRINT "I AM A GOOD COMUTER"
```

Do *not* press the **RETURN** key. Press the **DELETE** key several times until you have deleted everything after the **M**, and then type in **PUTER**". Press **RETURN** and your message will be displayed on the screen, thus:

```
I AM A GOOD COMPUTER
>
```

So far you have been using the computer in what is known as **immediate mode**. This means that when you type in an instruction the computer obeys the instruction when you press the **RETURN** key.

Putting a mistake right is known as **editing** it. Editing an incorrectly typed command is easily done by use of the **DELETE** key, which can be used to delete one character at a time. You will find that this key is useful in editing a line before you have pressed **RETURN**. If held down, however, it can rapidly delete a whole string of characters, so use it carefully.

It is often useful to be able to clear the whole screen. To do this type

in CLS and then press **RETURN**. Try it. Be warned, though: clearing the screen does not make the computer forget what you've already typed in—it just wipes out what is on the screen. If you want the computer to forget a program that you have made a mess of, press the **BREAK** key.

Now let's try to get the computer to obey more than one instruction. We want the computer to give the following greeting on the screen:

```
HELLO
WELCOME
```

As the computer can only read and obey one instruction at a time we will have to write what is known as a **procedure**. This will contain the instructions that we wish the computer to obey. The computer can then carry out the procedure, obeying the instructions one at a time.

Let us now type in our procedure, called, say, **GREETING**. Each line of instruction must be given its own line number. This makes it possible to refer to the line later, and, say, **LIST** lines 100 to 200, or change line 120, etc.

Type in the following lines, remembering to press **RETURN** after each one.

```
>100 DEF PROCGREETING
>110   PRINT "HELLO"
>120   PRINT "WELCOME"
>130 ENDPROC
```

DEF is short for 'define' and **PROC** is short for 'procedure', so instruction 100 is defining (giving the rules of) a procedure called **GREETING**. **ENDPROC** tells the computer that the procedure has ended. You will find a lot of abbreviations used in **BASIC**; this is mainly to save having to do too much typing. Shorter command words like **PRINT** are usually used in full. Where abbreviations are used, you will soon get used to them.

To get the computer to carry out the procedure type

```
>PROCGREETING
```

and press **RETURN**. The computer should give the following on the screen:

```
HELLO
WELCOME
>
```

If this does not happen, type **LIST** and then press **RETURN**. This lists the procedure you have typed in. Check each line with what you should

have typed in and retype any lines which are incorrect, along with their numbers. Then type **PROCGREETING** again.

In order to correct a mistake you can retype the whole of the line in which it occurs. If a line is typed into a program and it is discovered that the line is not needed it can be removed. Suppose that we had included a line numbered 115 in our program:

```
>115 PRINT "HI"
```

Such a line is removed by typing the number of the line and pressing the **RETURN** key.

```
>115      (just press RETURN)
```

Suppose you had made several mistakes while typing in your procedure **PROCGREETING**. It is all too easy to make typing errors when inputting your program. For example, try typing in this procedure.

```
>100 DF PROCGREETING
>110  PRINT "HELLO"
>120  PRUNT "WELCOME"
>130 END PROC
```

A quick glance suggests that everything is okay; but alas, there are three errors. Can you spot them?

On typing **>PROCGREETING** we get the response:

```
No such FN/PROC
>
```

The first error is that we have not defined the procedure—we typed **DF PROCGREETING** instead of **DEF PROCGREETING**. This error, at line 100, can be corrected by use of the edit keys **↑** **→** **←** and **↓** in conjunction with the **COPY** key.

Press the key labelled **↑** until the **edit cursor** sign **_** appears alongside line 100, as shown:

```
>LIST
_ 100 DF PROCGREETING
  110  PRINT "HELLO"
  120  PRUNT "WELCOME"
  130 END PROC
>
```

Now press the **COPY** key to copy one character at a time from the line 100 until you reach **D**, i.e.:

```
100 D
```


Now the missing character E can be typed, and then you can copy the remainder of the line. Finally, you must press the **RETURN** key to enter your corrected version of line 100 into the procedure already typed in. Now this new version of line 100 replaces the old one in the computer's memory.

Now try calling the procedure again before correcting the rest of the mistakes. Thus:

```
>PROCGREETING
```

The computer will respond:

```
HELLO
```

```
Mistake at line 120
```

```
>
```

The messages that the computer prints when it cannot obey your command, or cannot decide what to do, are not there to tell you off! They are called **reports**, or **error messages**, or **mishap messages**, and they often tell you exactly what the matter is and help you to put things right much faster. You can expect to encounter them regularly in your journey through computer land. As a matter of fact, they teach you a great deal. Therefore, consult the section on error messages in the *User Guide* and try to master the art of correcting the small errors which creep into your programs. These errors, by the way, are called **bugs** in computer jargon, and getting rid of them is called **debugging** a program.

Lines 120 and 130 can be corrected in a similar way, except that to avoid copying characters like the U in PRUNT, you must use the right-pointing arrows (**→**) to advance the cursor along the old line.

It may be helpful on occasions to isolate the incorrect line. To do this you may list the line concerned, e.g.:

```
>LIST 120                                (don't forget to press RETURN)
120 PRUNT "WELCOME"
```

Then you can edit the error much more easily.

Now type in a new procedure to say goodbye, called FAREWELL.

```
>300 DEF PROCFAREWELL
>310 PRINT "GOODBYE"
>320 ENDPROC
```

Type in

```
>PROCFAREWELL    and press RETURN
```

giving

```
GOODBYE
```

```
>
```

on the screen.

What we have done so far is write two procedures which the computer can use to give either a greeting or a farewell message. Now let us see how we can use these procedures to solve a problem. The problem is to get the computer to give a greeting message, draw a square and then give a farewell message. As we want this to be carried out automatically, we need to write a complete program that will carry out all the procedures for us.

The first step is to give our program a name, say **DRAWING A SQUARE**. We can do this by typing the BASIC keyword **REM** plus the program's name:

```
>10 REM DRAWING A SQUARE PROGRAM
```

REM is short for 'remark' and allows us to put helpful comments or remarks, such as a program title, within the program.

Now we need to run a procedure which will run all the other procedures. Let's call it **DRAWINGSQUARE**; so our next instruction is:

```
>20 PROCDRAWINGSQUARE
```

As yet we have not written the procedure **DRAWINGSQUARE**, but this is not important as we can always write it later. The next instruction is to tell the computer that the program is finished, so we type in:

```
>30 END
```

There we are, our program is written. To look at it, type

```
>LIST 10,30
```

and we get

```
10 REM DRAWING A SQUARE PROGRAM
```

```
20 PROCDRAWINGSQUARE
```

```
30 END
```

```
>
```

on the screen.

The next task is to write the procedure **DRAWINGSQUARE**. We need three procedures, one to give the greeting message, one to produce the square and the last one to give the farewell message. We have already written the first and the last, so we have only to write the middle one.

Let's call this procedure **SQUARE**. As before we need not write it until after we have written the procedure which uses it (**DRAWINGSQUARE**). The procedure **DRAWINGSQUARE** is therefore as follows. Type in:

```
>40 DEF PROCDRAWINGSQUARE
>50   PROCGREETING
>60   PROCSQUARE
>70   PROCFAREWELL
>80   ENDPROC
```

} the three procedures

Now we can write the procedure **SQUARE** by typing in the following:

```
>200 DEF PROCSQUARE
>210   PRINT
>220   PRINT "*****"
>230   PRINT "*   *"
>240   PRINT "*   *"
>250   PRINT "*****"
>260   PRINT
>270   ENDPROC
```

To run the program automatically, just type in

```
>RUN
```

and you will get:

```
HELLO
WELCOME
```

```
*****
*   *
*   *
*****
```

```
GOODBYE
>
```

To improve our greeting we can change **WELCOME** to **WELCOME TO A SQUARE**. Type in

```
>LIST 120
```

and on the screen will be displayed line number 120

```
120 PRINT "WELCOME"
>
```

Now retype this line as follows:

```
>120 PRINT "WELCOME TO A SQUARE"
```

Type RUN, and you get:

```
>RUN
HELLO
WELCOME TO A SQUARE
```

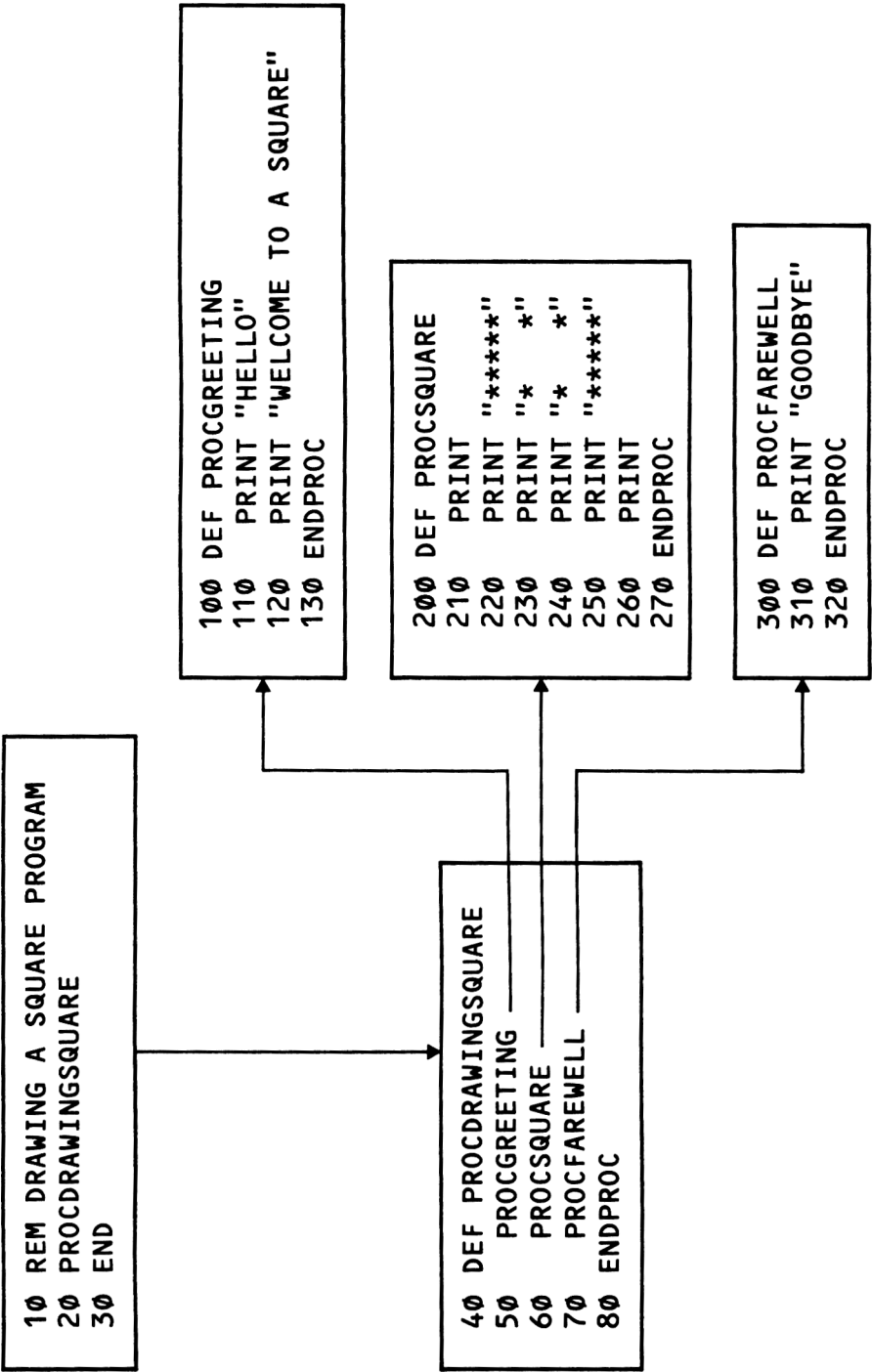
```
*****
*      *
*      *
*****
```

```
GOODBYE
```

Well done! You have now completed your first structured program in BASIC. To see the whole program, just type LIST. In this way you can see all of what you have programmed up to now.

```
>LIST
 10 REM DRAWING A SQUARE PROGRAM
 20 PROCDRAWINGSQUARE
 30 END
 40 DEF PROCDRAWINGSQUARE
 50   PROCGREETING
 60   PROCSQUARE
 70   PROCFAREWELL
 80 ENDPROC
100 DEF PROCGREETING
110   PRINT "HELLO"
120   PRINT "WELCOME TO A SQUARE"
130 ENDPROC
200 DEF PROCSQUARE
210   PRINT
220   PRINT "*****"
230   PRINT "*      *"
240   PRINT "*      *"
250   PRINT "*****"
260   PRINT
270 ENDPROC
300 DEF PROCFAREWELL
310   PRINT "GOODBYE"
320 ENDPROC
```

Fig. 1.2 The various parts of the DRAWING A SQUARE program.



Note that we have now used three different versions of the LIST command:

LIST with just one line number, to list only that line. For example—

```
LIST 120
```

LIST with two line numbers, to list between a lower limit and an upper limit. For example—

```
LIST 100,300
```

LIST with no line number, to list everything—

```
LIST
```

Now let us write a new program to draw a continuous rectangle on the screen. To clear the previous program from the computer, type NEW and then press **(RETURN)**.

Our rectangle program is:

```
>10 REM RECTANGLE PROGRAM
>20 PROCRECTANGLE
>30 END
```

And the rectangle procedure is as follows:

```
>100 DEF PROCRECTANGLE
>110 PRINT "*****"
>120 PROCRECTANGLE
>130 ENDPROC
```

Line 120 is what is called a **recursive** instruction, which means that the procedure is calling itself. Type RUN and see what happens.

Every time the computer gets to line 120, it re-enters the procedure and starts to obey the instruction on line 110. To stop the program, press the **(ESCAPE)** key.

Recursion can be a very powerful tool in programming, as you will find out later on. There is, however, a limit to the number of times the Electron BASIC can perform recursion. Therefore, if you didn't press the **(ESCAPE)** key to stop the PROCRECTANGLE 'chasing its own tail', it would stop with a message

```
No room
```

which means there is no more room for recursion.

Action keys introduced

RETURN	Enters typed line to computer.
BREAK	Restarts computer.
SHIFT	Allows use of upper row of characters where there are two alternatives. Also gives lower-case characters when the CAPS LK is on, and upper-case characters when the CAPS LK is off.
CAPS LK	Permits only upper-case alphabetic characters to be used at the keyboard.
ESCAPE	Stops the computer from obeying any more instructions until told to continue or restarted.
FUNC	Enables the BASIC keywords to be typed as recorded on the side front of the keys on the keyboard.

BASIC system commands introduced

CLS	Clears the TV screen.
RUN	Starts the program running.
LIST	Displays instruction lines on the screen.
NEW	Clears the present program.

Editing keys introduced

DELETE	Deletes one character to the left of the cursor.
COPY	Copies to the current line the character over the edit cursor.
	Moves the edit cursor one line up.
	Moves the edit cursor one line down.
	Moves edit cursor one character to the right.
	Moves the edit cursor one character to the left.

BASIC program commands introduced

PRINT	Displays characters on the screen.
DEF PROC	Defines procedure.
ENDPROC	End of procedure.
PROC	Procedure call.
REM	Allows remarks.
END	End of program.

Reflections

Ideally we would like to give instructions to a computer in English (or in any other language we happen to speak). On the other hand the machine is ideally suited to receive patterns of noughts and ones as that is how it is wired; it only knows what to do with patterns of noughts and ones. In the early days of computing, people had to give their instructions in **machine code** (the jargon phrase for strings of noughts and ones).

In order to bridge the gap, certain facilities are available on a computer to make it friendlier. We press keys which show the English alphabet. We are temporarily fooled into thinking that the computer understands English. Your early frustrations with the 'mistakes' the computer picked up has shown you this is not the case. Zero and O (the letter of the alphabet) mean two different things. A computer would not accept IO as ten. It would insist on 10. However, people would understand numbers written as IO, 2O, 3O, . . . Similarly, Figure 1.3 is clear as a meaningful sequence, both horizontally and vertically, to a human being; but it would thoroughly confuse a computer.

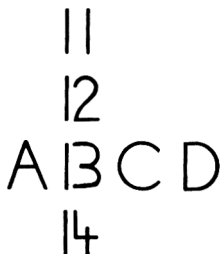


Fig. 1.3 A meaningful sequence to a human; not so to a computer.

In addition to all this, the computer performs several different tasks at the same time while you are composing your little program. In order to understand what goes on inside the computer, let us follow an analogy between recipe writing and program writing.

- (a) If my mother picks up a pen and a piece of paper to write a recipe in a language she understands (Persian) she is then like a programmer.
- (b) Having written the recipe she then posts it to me.
- (c) My job is to translate the recipe into English and pass it on to my English friend.
- (d) My friend can then follow the recipe and bake a cake.

Note that things can go wrong. If I (the translator) can't understand something, I have to send back my mother's letter with some questions about the bits I can't translate. Also, whilst cooking, my English friend could find out that my mother has forgotten something. She then has to ask me to tell my mother. . .

Writing a computer program, or getting something done by a computer, is similar to the job of my mother.

- (a) We need to use a facility called an **editor** to write out the instructions for doing a task in a language developed for talking to computers (structured BASIC in our case).
- (b) We then have to send these instructions to a translator (using what is known as an **Operating System**).
- (c) The job of the translator is to compile a translation of statements into the computer language in the actual code which a particular computer obeys internally, using a **compiler** or an **interpreter**.
- (d) The machine-code version of the program has to be put into the computer's memory, for the computer to follow.

The computer itself plays the role of all these mediators. It plays the 'hat-changing' game with the user (which can be rather confusing for newcomers).

BASIC makes life easier

Life would have been easier for my mother and my friend if they were both in the same place. My mother could have given the instructions one at a time. I could have interpreted them and my friend could have followed them. My mother could have watched in case anything went

wrong, sorting things out as we went along. There wouldn't have been any need for pen and paper or mailing of the recipe etc.

In the same way a programmer's life could be made easier if we could combine some pieces of the software and reduce the amount of mediation. This is what happens when you type a command without a line number. All of the jobs mentioned above get done immediately. Such commands, however, are not stored anywhere, and are lost after being obeyed.

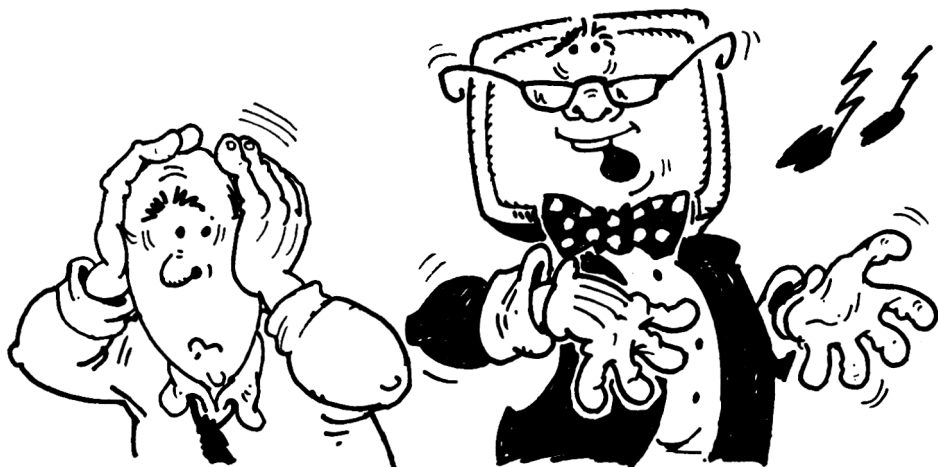
When we want to do complicated things we would rather write the instructions out, try them, and if necessary change them. The line numbering, the cursor-control keys, and the **(COPY)** and **(DELETE)** keys provide us with a basic editor. Through BASIC we can also access the Operating System of the computer. This means we can instruct the computer to read a file of programs into the memory and then to **RUN** it. These things have nothing to do with programming, but they do concern the 'operation' of the computer. In large computers the language for these different jobs is different. Also we have people employed as computer programmers and computer operators doing different jobs.

The fact that everything on your computer is done in BASIC makes life easier for the people who are beginning to learn about computers. They only need to learn about one beast rather than several.

This approach means that BASIC is 'Jack of all trades but master of none'. It also means that the full potential of the computer is obscured by the limits of BASIC. It is a toolkit consisting of one piece of machinery which is easy to learn to use but with which it is hard to produce masterpieces. Many other programming languages will become available for the Electron. Each has its own virtues.

2

Writing a Song



Ten Green Bottles

The program we are going to write will automatically produce the verses of the song *Ten Green Bottles*. First, let's type in the program.

```
>10 REM TEN GREEN BOTTLES PROGRAM
>20 CLS
>30 PROCTENGREENBOTTLES
>40 END
```

Now we can write the procedure TENGREENBOTTLES.

```
>100 DEF PROCTENGREENBOTTLES
>110   FOR I = 10 TO 1 STEP -1
>120     PRINT; I;" GREEN BOTTLES HANGING ON THE
        WALL,"
>130     PRINT; I;" GREEN BOTTLES HANGING ON THE
        WALL,"
>140     PRINT "AND IF ONE GREEN BOTTLE SHOULD
        ACCIDENTALLY FALL,"
>150     PRINT "THERE'D BE ";I-1;" GREEN BOTTLES
        HANGING ON THE WALL."
```

```

>160    PRINT
>170    NEXT I
>180    ENDPROC

```

We have already met the commands `DEF PROC`, `ENDPROC` and `PRINT`. The BASIC commands `FOR ... TO ... NEXT` form what is called a **loop** in computing. In line 110 you are asking the computer to count from 10 down to 1, and at each stage of the count to obey lines 120 to 160. The `NEXT` instruction tells the computer to go back to line 110 and continue the count.

The `STEP` command in line 110 allows us to count in steps of a certain size. When we omit the `STEP` from the `FOR ... TO ... NEXT` command, the step is taken by default to be 1. Here the step is -1: 1 is subtracted each time from `I`.

The program refers several times to `I`. This is a **variable**, something which can vary its meaning during the running of a program. In actual fact `I` is the **name of a variable**. Think of `I` as the name of a box (variable) which can hold only one number (value) at a time. During the course of the running of this program, the box called `I` holds several different numbers, but of course it only holds one at a time. For example, 10 is the first number it holds, and 1 is the last.

Thus, as indicated in Figure 2.1,

- a **variable** is likened to a box;
- its **name** is likened to the name of the box;
- its **value** is likened to one and only one number in the box.

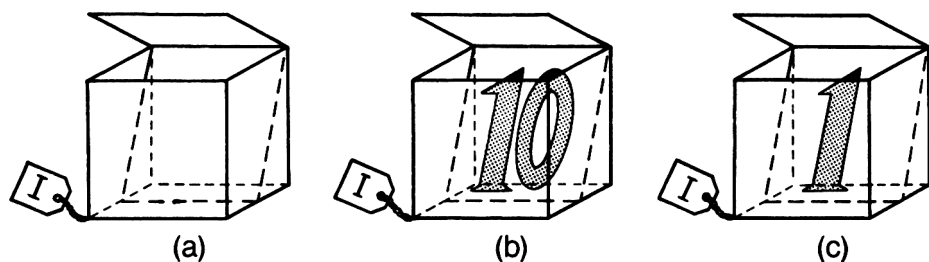


Fig. 2.1 A picture of the variable `I`
 (a) before the first verse;
 (b) during the first verse;
 (c) during the last verse.

Variables can be created as a result of a `LET` command in a program, or when we use them inside a loop. We could write:

```

>105 LET I=10

```

In lines 120, 130 and 150 we use the current value in the box labelled I to give us the correct number of bottles for the particular verse. The computer will display the contents, or value, of I, because we have not used quotation marks either side of the I. So with the PRINT command you can either print the contents of a variable box, or, by using quotation marks, print a series of characters. The semicolons tell the computer that we do not wish to print any spaces between these two parts of the line we are printing.

Now type in RUN to see the result of this program. You should get the following:

```
10 GREEN BOTTLES HANGING ON THE WALL,
10 GREEN BOTTLES HANGING ON THE WALL,
AND IF ONE GREEN BOTTLE SHOULD ACCIDENTALLY
FALL,
THERE'D BE 9 GREEN BOTTLES HANGING ON THE
WALL.
```

```
      :
      :
      8 verses
      :
      :
```

```
1 GREEN BOTTLES HANGING ON THE WALL,
1 GREEN BOTTLES HANGING ON THE WALL,
AND IF ONE GREEN BOTTLE SHOULD ACCIDENTALLY
FALL,
THERE'D BE 0 GREEN BOTTLES HANGING ON THE
WALL.
```

It would be nice if we could end the last verse as follows:

```
THERE'D BE NO GREEN BOTTLES HANGING ON THE
WALL.
```

To do this we will have to find some way of testing in line 150 whether we are on the last verse or not. We can do this by using the command IF ... THEN ... ELSE ..., as follows. Note that all of line 150 must be typed in as one line, without pressing (RETURN) until the end of it. On your screen its spacing will probably not look the same as it does here.

```
>150 IF I=1 THEN PRINT "THERE'D BE NO GREEN
      BOTTLES HANGING ON THE WALL."
      ELSE PRINT "THERE'D BE ";I-1;" GREEN
      BOTTLES HANGING ON THE WALL."
```

Try typing in the above line and then type RUN to see that it works. Note that all of that long IF ... THEN ... ELSE ... statement has to be

typed as one BASIC line, i.e. it must have the same line number and be typed with **(RETURN)** pressed only at the end. This, however, might look like several lines on the TV screen.

What is happening is that the computer tests the condition 'is I equal to 1?'. If it is, we are on the last verse and can print *the last line of the song*. If I is not equal to 1, then we need to print *the last line for that verse*. This is called **conditional branching** and it can be very useful in computer programming.

You may like to save a copy of your program for future use. To do this, think of a name for it (up to ten characters), say **GREEN**. Then use the **SAVE** command thus:

```
>SAVE "GREEN" (RETURN)
```

The computer will reply by:

```
RECORD then RETURN
```

Press the **RECORD** button on your cassette player and then press the **(RETURN)** key. When the computer has finished saving your program, it will beep and the prompt sign, **>**, will reappear on your screen.

If you switch off the computer now, the version of your program in the computer's memory will be lost but the copy on the cassette is safe. Rewind your tape for the time you want to load it later.

When you switch on the computer again you can load the program using the **LOAD** command.

```
>LOAD "GREEN" (RETURN)
```

Don't forget to press the **(RETURN)** key! Put the cassette in your cassette player then press the **PLAY** button on your cassette recorder. When you do this the computer will wait for a while and then click quietly, at the same time writing on the screen:

```
Loading
GREEN 00
```

This means that it has found the first part of the program. The program is recorded on the tape in small pieces, called **blocks**, as this makes it easier for the computer to check that it is reading the tape properly. The double zero is called the **block number**; every few seconds this will increase by one, to show that the next block has been found and is being loaded. The numbers actually count in something called **hexadecimal**, which means that after 9 appears the letters A to F are also used; see the *User Guide* if you want to know more about this. When the last block has been found and read, a new four-digit number will appear beside the

block number, but you can ignore that for the time being.

If you get any other messages on the screen you may have to reset the volume control on your cassette player. It is also possible that the program has not loaded properly onto the cassette. For instructions see the *User Guide*.

After a short time the computer will beep at you, which is its way of telling you that it has loaded the program for you, and give you the familiar prompt sign >. Now check to see if this has happened, either by typing

>RUN (RETURN)

or by pressing (FUNC) key and (R) key together. If you do not get the outcome expected, list your program and, if there are problems, ask somebody to check your cassette player for you.

There are many ways in which the TEN GREEN BOTTLES program could have been written; try writing a new version of your own, which is different from the one above. You could, for example, start by tidying up the program so that the layout of the song is better. You could also make use of lower-case characters. In Chapter 1 you were advised to keep the (CAPS LK) on, so whenever you press a key you get the upper-case version. However, while (CAPS LK) is on you can get the lower case by pressing the (SHIFT) key and the character key you want (i.e. (G) to get g).

Also note that all of line 150 must be typed in as one line, without pressing (RETURN) until the end of it. On your screen its spacing will probably not look the same as it does here.

```
> 10 REM *** A modified version of "TEN GREEN
    BOTTLES" ***
> 20 CLS
> 30 PRINTTAB(15) "TEN GREEN BOTTLES"
> 40 PRINTTAB(15) "=====
> 50 PRINT:PRINT
> 60 PROCTENGREENBOTTLES
> 70 END
>100 DEF PROCTENGREENBOTTLES
>110   FOR I=10 TO 1 STEP -1
>120     PRINT; I; " green bottles hanging on
        the wall,"
>130     PRINT; I; " green bottles hanging on
        the wall,"
>140     PRINTTAB(10) "And if one green bottle
        should accidentally fall,"
```

```

>150    IF I=1 THEN PRINTTAB(10)
        "There'd be no green bottles
        hanging on the wall."
        ELSE PRINTTAB(10)
        "There'd be "; I-1;" green bottles
        hanging on the wall."
>160    PRINT
>170    NEXT I
>180    ENDPROC

```

PRINTTAB is another form of printing on the screen which enables you to specify the number of leading spaces.

Exercises

1 These programs all produce 1 GREEN BOTTLES for their last verse. Modify them to avoid the grammatical error.

2 Write a program for the song *Ten in the Bed*, which starts:

```

THERE WERE 10 IN THE BED,
AND THE LITTLE ONE SAID,
'ROLL OVER, ROLL OVER.'
SO THEY ALL ROLLED OVER,
AND ONE FELL OUT.
THERE WERE 9 IN THE BED,
...

```

3 Try the song about mowing a meadow:

```

1 MAN WENT TO MOW,
WENT TO MOW A MEADOW.
1 MAN AND HIS DOG WENT TO MOW A MEADOW.

2 MEN WENT TO MOW,
WENT TO MOW A MEADOW.
2 MEN, 1 MAN AND HIS DOG WENT TO MOW A MEADOW.

3 MEN WENT TO MOW,
...

```


Action keys used in this chapter—a reminder

RETURN	Enters typed line to computer.
SHIFT	Allows use of upper-case characters when CAPS LK is off. Or allows use of lower-case when CAPS LK is on.
CAPS LK	When switched on, makes upper-case characters the norm, and when switched off makes lower case the norm.
FUNC	Enables the BASIC keywords to be typed as recorded on the side front of the keys on the keyboard.

BASIC system commands introduced

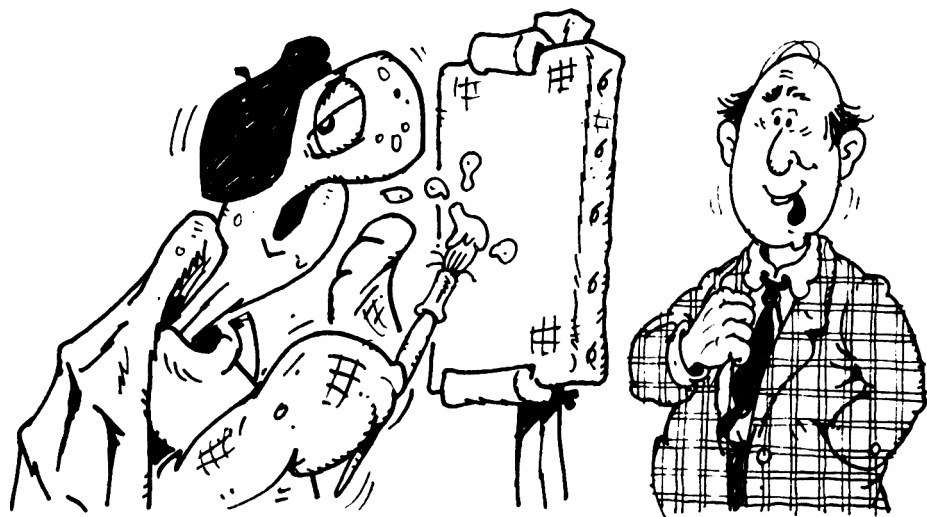
SAVE	Saves, on the cassette, a copy of the program in the computer's memory.
LOAD	Loads a program from the cassette into the computer's memory.

BASIC program commands introduced

FOR...TO... STEP...NEXT... (STEP	A loop using a count to determine the number of times to repeat the loop. introduces the step size for the count in the FOR loop.)
IF...THEN... ELSE...	A conditional branching command.
PRINTTAB	Displays characters on the screen with a given number of leading blanks.
LET...=...	Assigns a value to a variable (puts a number into a box with the given name).



Talking to Turtle



The world of Turtle

This chapter introduces you to the world of a computer creature known as 'Turtle'. It is a friendly creature which can draw pictures for you on your TV screen. It is similar to drawing with a pen on a piece of paper. Here, it is Turtle who is holding the pen, and you order Turtle rather than your hand to do the drawing. Turtle understands several different commands. In this unit we introduce you to some of the most important of Turtle's commands.

First, to see the world in which Turtle is living, connect up the cassette recorder as described in the *User Guide*, insert side B of the *Introductory Cassette*, and give the command:

>LOAD "TURTLE"

Don't forget to press the **RETURN** key! Now rewind the cassette to the beginning of side B (note that if your cassette recorder has motor control, you will not be able to rewind until you have issued the **LOAD** command). Then press the **PLAY** button on your cassette recorder.

When you do this, the computer will first write on the screen

Searching

then

Loading

TURTLE 00

If you do not happen to have a copy of the cassette with the programs on it, you can use the listings of them in Appendix A at the back of this book and type them in yourself. However, be prepared for error messages when you first run the programs. You will probably make a few typing mistakes (such as typing 0 instead of Ø, or missing a line or two). Ask somebody to check each program before you use it. In Chapter 2 you learnt how to save programs on a cassette, so when you have had your program corrected you can save it for future use.

When the last block has been found and read, a new four-digit number will appear beside the block number, but you can ignore that for the time being. After a short time the computer will beep at you, which is its way of telling you that it has finished loading the program. STOP THE TAPE IMMEDIATELY if you do not have motor control. Type:

>RUN

Again, don't forget to press the **RETURN** key!

You'll see a map of Turtle's world with a lone character ↑ inside it, as Turtle's world is empty to begin with. However, when you get Turtle moving it will leave a trace of 'paint' behind it.

Turtle's world can be thought of as made out of little squares. The initial location where Turtle is resting is called (1,1). Turtle starts off in the bottom left corner, facing north. Type:

>PROCFORWARD(5)

This tells the Turtle to move five units forward.

Turtle can be turned to face in one of eight directions—up, down, left, right, or one of the diagonals—by telling it to turn to the left or right using **PROCLEFT** and **PROCRIGHT**. These directions are labelled like a compass, as shown in Figure 3.1. You are always told which direction

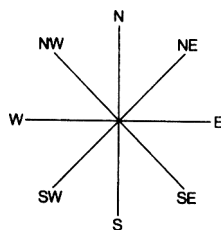


Fig. 3.1 Turtle can be turned to face in each of these eight directions.

Turtle is currently facing, on the line just below the picture, where you are also told Turtle's current position in its world. Because each of these directions is a 45-degree turn from its neighbours, you can only tell Turtle to turn by a multiple of 45 degrees, i.e. 45, 90, 135 and so on. If you try anything else, Turtle will complain. So, try typing:

```
>PROCLEFT(17)
```

Now try:

```
>PROCRIGHT(90)
>PROCFORWARD(5)
>PROCRIGHT(90)
>PROCFORWARD(5)
>PROCRIGHT(90)
>PROCFORWARD(5)
```

You should have a square in the picture, and Turtle facing west. If you give the command

```
>PROCFORWARD(5)
```

you'll see that Turtle can't break out of its world! Let's get Turtle facing north again, by typing:

```
>PROCRIGHT(90)
```

Now let's try to draw another square. Instead of typing the same sequence of `PROCFORWARD(5)`, `PROCRIGHT(90)` we can use a command which tells the Turtle to repeat something a number of times. So let's teach Turtle a new command for drawing a square:

```
>6000 DEF PROCSQUARE
>6010   FOR C = 1 to 4
>6020     PROCFORWARD(5)
>6030     PROCRIGHT(90)
>6040   NEXT C
>6050 ENDPROC
```

Note that we have written our program from line 6000 onwards as, by loading the tape, we have filled the computer's memory up to line 6000 with the definitions of Turtle programs.

Now try:

```
>LIST 6000,6050
```

The last line will list what you have just typed in, but most of it will have

disappeared before you can read it, as there is space for only a few lines of text available at the moment on the screen. You could list each line on its own and correct it, using the cursor and **(COPY)** keys, by typing

```
>LIST 6000
```

to see line 6000, or LIST 6010 to see line 6010 and so on. However, you will find it easier if you type

```
>MODE 6
```

```
>LIST 6000,6050
```

which will clear the screen and let you see all the lines of the procedure at once. When you think the procedure has been typed in properly, type:

```
>RUN
```

When you have typed the RUN command, all you will see happen is the screen going blank, and the Turtle's world being redrawn, exactly as it was when you typed RUN for the first time after loading the program. This is because when you add something to the program, or make a correction to something you mistyped, the computer forgets that the Turtle is there, even though you can still see it on the screen. When it redraws Turtle's world, the Turtle is put back in the starting position, which the computer does know about.

We have already met FOR...TO...STEP...NEXT. When we omit the STEP and write FOR...TO...NEXT, the step is taken by default to be 1. In line 6010 you are asking the computer to count from 1 to 4, and at each stage of the count to obey lines 6020 and 6030. The NEXT instruction tells the computer to go back to line 6010 and continue the count. The effect is to perform the two procedures FORWARD and RIGHT four times.

Before trying the procedure to see if it has learned about square drawing, you might wish to make Turtle leave a different sort of trail when it draws. If you wish to make it leave a trail of dots, instead of asterisks, try:

```
>PROCCHANGEPAINT(".".)
```

Now try calling the procedure you have just defined, to get it to do something for you.

```
>PROCSQUARE
```

The only problem is that however many times you tell it to draw a square, it draws one of size 5! So let's enable our procedure to draw a square of any size, by changing lines 6000 and 6020 to:

```
>6000 DEF PROCSQUARE(SIZE)
>6020   PROCFORWARD(SIZE)
>RUN
```

The **SIZE** can be thought of as a label put on a box which is in the computer. We can place any value in this box. When operating the procedure **FORWARD**, the computer looks into the box labelled **SIZE** and uses the value for the number of places to move forward.

Now we can make our procedure draw a square of any size. Try the following sizes of squares.

```
>PROCCHANGEPAINT("+")
>PROCSQUARE(5)
>PROCSQUARE(7)
>PROCSQUARE(9)
```

You might want to clear the screen and start all over again. To do this, use **PROCCLEARSCREEN**. This will erase the picture you have drawn, and put Turtle at the starting position. However, it *won't* make Turtle forget what you have taught him.

You might like to draw pictures in a different area of Turtle's world. You have to tell Turtle to take up his pen from the picture, and then move to where you would like him to go. Try:

```
>PROCPENUP
>PROCRIGHT(45)
>PROCFORWARD(5)
>PROCPENDOWN
```

Now type:

```
>PROCSQUARE(5)
```

Try typing this short procedure, and see what happens:

```
>6060 DEF PROCSTAR(SIZE)
>6070   FOR C=1 TO 8
>6080     PROCFORWARD(SIZE)
>6090     PROCRIGHT(180)
>6100     PROCFORWARD(SIZE)
>6110     PROCLEFT(45)
>6120   NEXT C
```

```
>6130 ENDPROC
```

```
>RUN
```

Move the Turtle to the middle of the screen, and try:

```
>PROCSTAR(5)
```

If you have ever had the message

```
No such variable at line ...
```

with a number less than 6000, it means that you have forgotten to type RUN after making a change to the program. If you use a blank screen when you want to change the program then you shouldn't forget to do this, as otherwise you can't see the Turtle on the screen.

Try stars of different sizes. Draw them in different places. If you think the screen is becoming crowded, type:

```
>PROCCLEARSCREEN
```

When you have done this, try typing in:

```
>PROCPENUP  
>PROCRIGHT(45)  
>PROCFORWARD(10)  
>PROCPENDOWN  
>PROCSTAR(6)  
>PROCCHANGEPAINT(" ")  
>PROCSTAR(6)
```

You can use the procedures you have already defined when writing a new one. To show this, think about what this procedure draws, and then try it on the computer:

```
>6140 DEF PROCFOURSQUARE(SIZE)  
>6150   FOR D=1 TO 4  
>6160     PROCSQUARE(SIZE)  
>6170     PROCRIGHT(90)  
>6180   NEXT D  
>6190 ENDPROC  
>RUN  
>PROCPENUP  
>PROCRIGHT(45)  
>PROCFORWARD(10)  
>PROCLEFT(45)  
>PROCPENDOWN  
>PROCFOURSQUARE(4)
```

Again, try this in different places on the screen, and with different sizes.

Now, as a change, have a go at drawing a stick figure of a man, giving him a head, a body, arms and legs. If you draw him on the screen, you can then undraw him by drawing over him using blanks, as you did with the star. After you have managed this, move along the screen a bit, and draw him again. By doing this a few more times, you can make your man walk across the screen.

To make this easy, you should first write a procedure to draw the man, then try to undraw him again by using the same procedure, but with a different paint, a blank. So that you can save some time while typing in all those lines, you can use short names for the Turtle commands. The following list shows all the commands you can use, with both their long and short names.

Turtle procedures

<i>Full name</i>	<i>Short name</i>	
PROCCLEARSCREEN	PROCCS	Clears the screen in which Turtle lives.
PROCFORWARD	PROCFD	Moves Turtle forward.
PROCRIGHT	PROCRT	Turns Turtle to the right.
PROCLEFT	PROCLT	Turns Turtle to the left.
PROCPENUP	PROCPU	Takes the pen up from the screen so that Turtle will not leave a trail behind.
PROCPENDOWN	PROCPD	Puts the pen down so that Turtle will leave a trail behind after its movement.
PROCCHANGEPAINT	PROCCP	Changes the paint used for the trail left behind Turtle.

For more about Turtle Graphics and the LOGO philosophy of education, see *Mindstorms – Children, Computers and Powerful Ideas* by Seymour Papert (Harvester Press, Brighton, 1981.)



Playing with Numbers



What's in a name?

So far we have used numbers to count the number of times we do something. We have put numbers into what we referred to as boxes, to store them; and added to numbers in specified steps when we used the NEXT part of FOR . . . TO . . . STEP . . . NEXT. In the last chapter we chose I as a name for the box into which we could put a number. Examples of such names are:

I N J

We can, however, write long meaningful names in Electron BASIC, so long as we make sure that these do not begin with names already used as part of BASIC. A name must always start with a letter, and must not have a space or punctuation mark in it. We can use upper-case or lower-case letters, and we can use the underline character to make two-word names readable.

Let's try working out the area of a rectangle. We can do this by writing a procedure to calculate an area, say 5 units by 6 units. Type in the following:

```
>10 DEF PROCAREA  
>20 LET length = 5
```

```

>30 LET width = 6
>40 LET area = length * width
>50 PRINT "AREA IS "; area
>60 ENDPROC

```

Now type:

```
>PROCAREA
```

You should get the following:

```
AREA IS 30
```

In line 20 we have defined a box (a variable) called `length` and put 5 in it. In line 30 we have defined a box called `width` and put 6 in it. In line 40 we have defined a box called `area` and put the result of the multiplication of `length` by `width` in it, using the BASIC multiplication operator, which is `*`. In line 50 we then print the value currently in the box called `area`.

We could not have used `LENGTH` (upper case) as it would have been confused with the BASIC word `LEN`.

These names have all been used to stand for whole numbers, or integers. Numbers themselves can be used in BASIC, and it is fairly obvious what they mean. We can also put fractional numbers into boxes, using the full stop as a decimal point.

Simple expressions

An **expression** is a rule (or formula) for calculating a value from the values of the parts which it is made out of. There were several obvious examples of expressions in the earlier chapters, e.g.:

$$K = I - 1$$

Expressions are evaluated each time they are encountered in the execution of a program. Whatever the value of `I` is when the above expression is encountered, the value of the expression will be one less.

There are some simpler expressions we have used, such as

$$10 \quad I \quad J \quad 4$$

where there is no need for the calculation of values.

How to write expressions

The expressions in BASIC are intended to follow closely the formulae of mathematics, whilst at the same time being unambiguous when written out as a linear sequence of characters.

Expressions are made up from:

operators (e.g. +, -)

operands (e.g. identifiers, numbers)

brackets

The principal differences between the formulae of mathematics and expressions in BASIC are:

- (a) All multiplication must be written out and is denoted by * and not $\times$.
- (b) The mathematical expression $\frac{I}{J}$ (or $I \div J$) is represented in BASIC by I/J.

Using LET and =

We have defined a variable as an object which has a name and a value. We have only described one rather special way of changing the value of a variable, namely the repetitive FOR statement. In Electron BASIC, the **assignment statement** provides a simple way of putting new values into existing variables. For example,

LET I = 13 is read as 'let I become thirteen'.

To the left of the = sign, there must be the name of the receiving variable. To the right of the = sign, there must be an expression whose value is put into the variable named on the left. We can omit the word LET in Electron BASIC. For example:

J = I + 1

In many cases the expression on the right-hand side is an extremely simple one, e.g.

>J = 0

>I = J

>B = 3

One interesting situation is when we write:

I = I + 1

What we are doing is adding one to the old value of I and making it the new value of I. Now what is the precise meaning of the assignment statement?

- (a) the expression on the right-hand side is evaluated;
- (b) this value is put into the variable named on the left-hand side.

We can now understand the meaning of:

$K = K + 1$	add 1 to the value of K
$J = J * 2$	double the value of J
$I = (J - 1) * (J + 1)$	replace I by $(J - 1)(J + 1)$ or $J^2 - 1$!

Let us try another calculation of area, this time of all the rectangles with a length of 5 and width a whole number of units, till we find a rectangle with an area of 80. First of all type:

>NEW

This tells the system you want to forget the old stuff you have written and you are going to write something new. Now we can type the new version of PROCAREA:

```
> 10 DEF PROCAREA
> 20   LET length = 5
> 30   LET width = 1
> 40   REPEAT
> 50     LET area = length * width
> 60     PRINT "WHEN WIDTH "; width
> 70     PRINT "AREA IS "; area
> 80     LET width = width + 1
> 90   UNTIL area=80
>100 ENDPROC
```

Type the following:

>PROCAREA

REPEAT . . . UNTIL . . . is for repeating a set of commands, each time checking if a condition has been satisfied. If 'yes' (or in computer terms the condition is TRUE) then we stop. If 'no' (or in computer terms the condition is FALSE) we carry on. In this case we are checking if area is 80 or not. In BASIC we can check a number of conditions using inbuilt operators such as > (for greater than) < (for less than) and <> (for not equal). We can also combine several conditions by using AND, OR and

NOT, which are briefly described in the glossary and in detail in the *User Guide*.

Exercise

Write a program which calculates the area of squares with an area of up to 100.

Functions

After playing with numbers for a while you may start to need to do more sophisticated things with them than multiplication and division. Electron BASIC has a number of useful inbuilt functions, such as one which works out the square root of a number. Try:

```
>PRINT SQR(81)
```

The full list of Electron BASIC functions is in the *User Guide* and you should try and discover some of them. One very useful function is for generation of random numbers. **RND(X)** generates a random number between (and including) 1 and X. Try the following:

```
>NEW
>10 DEF PROCSTRANGE
>20 REPEAT
>30   FOR I=1 TO RND(35)
>40     PRINT "*";
>50   NEXT I
>60 PRINT
>70 UNTIL I>35
>80 ENDPROC
```

Now call the procedure several times:

```
>PROCSTRANGE
>PROCSTRANGE
>PROCSTRANGE
```

Each time you should get a random picture, but all of them have 35 stars in their last line.

In addition to **inbuilt functions** there is a BASIC structure which makes it possible for us to write our **own functions**. Try the following:

```

>NEW
>100 DEF FNPOWER(X,Y)
>110   LOCAL I,J
>120   LET I=1
>130   LET J=Y
>140   REPEAT
>150     LET I=I*X
>160     LET J=J-1
>170   UNTIL J=0
>180 =I

```

Now try:

```

>PRINT FNPOWER(2,3)
>PRINT FNPOWER(3,4)
>PRINT FNPOWER(2,8)

```

The power is calculated here as a series of multiplications. We store the initial value of Y in J and keep taking 1 off J while multiplying our total (saved in I) by X. On line 100 we define a new function called POWER and on line 180 we say that the result of power is I. In line 110 LOCAL is a BASIC command which defines variables as local work space for the use of the function. When we have finished with the function we lose the values of I and J. Only the result, after the =, is left after a call to the function.

Note that the power function calculates a value: it is not like a procedure to be used on its own, so

```
>FNPOWER(2,8)
```

would not work. Try it.

BASIC system commands introduced

NEW	Removes a program from the computer's active memory to enable you to write a new program.
OLD	Recovers the old program recently deleted.

BASIC program commands introduced

REPEAT ... UNTIL ...	Repeat a sequence of statements until a condition becomes true.
DEF FN	Define a new function.
= ...	Finish the function and return the value given.
LOCAL	Define variables as local work space inside a function.

BASIC inbuilt functions introduced

RND(X)	Generates a random number between 1 and X.
SQR(X)	Return the square root of X.

BASIC inbuilt operators introduced

Note: Not all of these were introduced in Chapter 4, but it would be useful for you to experiment with them.

... + ...	Add two numbers.
... - ...	Subtract two numbers.
... / ...	Divide two numbers.
... * ...	Multiply two numbers.
... ^ ...	Raise the first number to the power of the second.

... = ...	Gives TRUE if the two numbers are equal.
... > ...	Gives TRUE if the first number is greater than the second.
... < ...	Gives TRUE if the first number is less than the second.
... <> ...	Gives TRUE if the two numbers are not equal.
... AND ...	Joins two conditions together to make a new one that is TRUE if both the first and the second conditions are TRUE .
... OR ...	Joins two conditions together so that the whole is true if one, or the other, or both of the conditions are TRUE .
NOT ...	Negates a condition, so that the result is TRUE if the condition is FALSE , and <i>vice versa</i> .

5

Structured Problem Solving



Introduction

We studied the **TEN GREEN BOTTLES** program to discover how it worked. In this chapter we are interested not only in the program, but also in how we derive it. We use the method of **top-down refinement**, or what is known as **structured programming** in Computer Science departments!

The problem is to draw an isosceles triangle (one with two of its sides equal) made out of asterisks (stars), as shown in Figure 5.1. We should

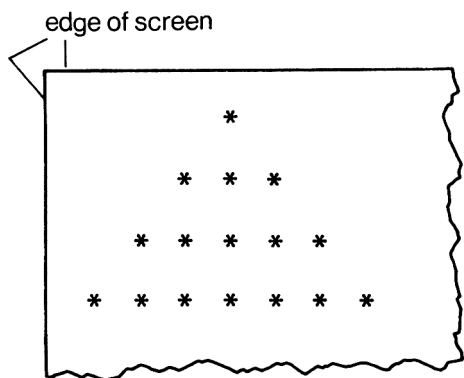


Fig. 5.1 An isosceles triangle made out of asterisks.

be able to recognize this 4-line triangle as a special case of a triangle of many lines (which we call an N -line triangle). Let us suppose that BASIC had a statement `PROCDRAWTRIANGLE(N)`. If this were so, the program would simply be:

```
10 REM ISOSCELES PROGRAM
20 PROCDRAWTRIANGLE(4)
30 END
```

But as this `PROCDRAWTRIANGLE` is not a BASIC statement, we must introduce a new command by means of a procedure declaration. That is, we must:

- (a) give it a name—obviously `PROCDRAWTRIANGLE`;
- (b) say what it means.

We now have to think about how to draw the triangle on a device which is designed for typing text. Clearly we have to draw the triangle out a line at a time, from top to bottom. Let us suppose that BASIC has a statement `PROCDRAWLINE(I)` which draws the I th line of an N -line triangle. Our program would be:

```
> 5 CLS
>10 REM ISOSCELES PROGRAM
>20 PROCDRAWTRIANGLE(4)
>30 END
>40 DEF PROCDRAWTRIANGLE(N)
>50   FOR I = 1 TO N
>60     PROCDRAWLINE(I)
>70   NEXT I
>80 ENDPROC
```

Now there are two names used in this program which have not yet been introduced and explained. We should immediately recognize `PROCDRAWLINE` as one of these, for we deliberately supposed it to be known to BASIC. The other one is the variable I ; we use this to count the lines as we draw them, and to indicate which line is being drawn at each stage by `PROCDRAWLINE`.

Now for the procedure `DRAWLINE`. Here we come to the real heart of the problem. How do we draw a given line, say the K th line of an N -line triangle? Table 5.1 shows the rules when N is 4.

As we want to find a general way of drawing isosceles triangles of any size, we have to find a table for an N -line triangle. Table 5.2 shows the relationship between the line number K and the size N .

Table 5.1

Line number	Leading spaces	Stars					
1	3	1				*	
2	2	3			*	*	*
3	1	5		*	*	*	*
4	0	7	*	*	*	*	*

Table 5.2

Line number	Leading spaces	Stars	What to do at the end of the line
1	$N-1$	1	PRINT
2	$N-2$	3	PRINT
3	$N-3$	5	PRINT
:	:	:	:
K	$N-K$	$2K-1$	PRINT
:	:	:	:
$N-1$	1	$2N-3$	PRINT
N	0	$2N-1$	PRINT

We are now left with the problems

(a) how to output a space: `PRINT" ";`

(b) how to output a star: `PRINT"*";`

and then

(c) how to output $N-K$ spaces:

```
>100 FOR J = 1 TO N-K
>110   IF K<N THEN PRINT" ";
>120 NEXT J
```

The IF $K < N$ is because we only want spaces on lines before the last one (the N th one). On the last line we don't want to print any spaces.

(d) how to output $2K-1$ stars:

```
>130 FOR J = 1 TO 2*K-1
>140   PRINT"*";
>150 NEXT J
```

Note: (a) The symbol for multiplication is $*$.

(b) Multiplication signs may *not* be omitted as they may in algebra.

We can now write the procedure **PROCDRAWLINE**.

```
> 90 DEF PROCDRAWLINE(K)
>100   FOR J = 1 TO N-K
>110     IF K<N THEN PRINT" ";
>120   NEXT J
>130   FOR J = 1 TO 2*K-1
>140     PRINT"*";
>150   NEXT J
>160   PRINT
>170 ENDPROC
```

This procedure contains no unknown statements; it is entirely in proper BASIC statements. Our task is almost finished. The complete program is:

```
>LIST
  10 REM ISOSCELES PROGRAM
  20 PROCDRAWTRIANGLE(4)
  30 END
  40 DEF PROCDRAWTRIANGLE(N)
  50   FOR I = 1 TO N
  60     PROCDRAWLINE(I)
  70   NEXT I
  80 ENDPROC
  90 DEF PROCDRAWLINE(K)
 100   FOR J = 1 TO N-K
 110     IF K<N THEN PRINT" ";
 120   NEXT J
 130   FOR J = 1 TO 2*K-1
 140     PRINT"*";
 150   NEXT J
```

```
160 PRINT
170 ENDPROC
```

>RUN

Now, in order to see the results of your hard work, try a few direct commands to see if your program can produce triangles of different sizes:

```
>PROCDRAWTRIANGLE(6)
>PROCDRAWTRIANGLE(15)
```

Exercises

- 1 Modify the program to print the triangle upside down.
- 2 Modify the program to leave a left-hand margin of 6 spaces.
- 3 Modify the program to leave a left-hand margin of N spaces.
- 4 Modify the program to draw a right-angled triangle, thus:

```

      *
    * * *
  * * * * *
* * * * * *
```

- 5 Modify the program to draw a right-angled triangle, thus:

```

                        *
                      * * *
                    * * * * *
                  * * * * *
                * * * * * *
```

- 6 Produce a program to draw:

```

* * * * * * *
* * * *
* * *
*
```

Further exercises

You should move to the next chapter and come back to these exercises later, unless you have found the earlier exercises too easy!

7 Produce a program to draw a diamond:

```

      *
    * * *
  * * * * *
* * * * * * *
  * * * * *
    * * *
      *
  
```

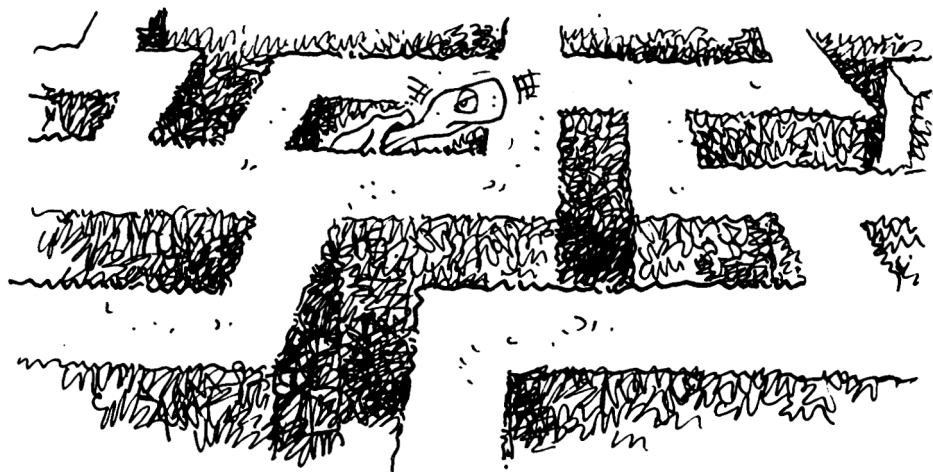
8 Produce a program which draws a diamond with asterisks only on its perimeter, thus:

```

      *
    *   *
  *     *
*       *
  *     *
    *   *
      *
  
```



Helping Turtle in a Maze



Giving Turtle senses

In Chapter 3 you were introduced to a fictional animal which can take your orders and produce pretty pictures for you. Let's pursue the analogy of Turtle as an animal a step further, by thinking of his movements as a behaviour pattern, and of the Turtle programs as models of simple animal behaviour.

So far, we have been introduced to primitives by which a complex pattern of movement can be performed. We can make our Turtle perform more 'intelligent' tasks by allowing its behaviour to be affected by some sensory data from its world. So far we have made Turtle start from a world where nothing exists. However, if we use an old picture with something already in it, we need to give Turtle senses which will help him 'understand' the picture.

The Turtle package has several functions which give useful values. **FNXPOS** and **FNYPPOS** give the current X and Y coordinate positions of the Turtle; **FNHEADING** gives its current heading (direction). These three functions all, in a way, describe the 'state' of the Turtle. **FNUNDER**, however, is like a sensory function; it gives the content of the point in the picture on which Turtle is standing.

Before trying out these new functions, there is something you should note about using the Turtle package. When you press the **ESCAPE** key to stop a program, it can sometimes leave the cursor in Turtle's world. If this does happen you must **RUN** the program again before doing anything else; but don't forget that this will clear away the picture, so make a note of what was going on to make you press **ESCAPE** in the first place.

Turtle in a maze

With the *Introductory Cassette* tape positioned as it was immediately after loading **TURTLE**, type

```
>LOAD "SOLVER" RETURN
```

and press the **PLAY** button on the cassette recorder. If you don't get a Loading message (which might happen, for example, if you did not stop the tape quickly enough after loading **TURTLE**), just rewind the tape a little way, then press **PLAY** again.

When the program has finished loading (you will hear a beep) **STOP THE TAPE IMMEDIATELY** if you don't have motor control, then type:

```
>RUN followed by RETURN
```

Now, using the functions mentioned above, we can get Turtle into a maze and help it to find its way from the start to the finish. You can get Turtle to draw your own maze and use that, but for the moment you can use the **PROCGETMAZE** procedure in the Turtle package. Try typing:

```
>PROCGETMAZE(1)
```

Number 1 means you want the first maze in the set of 7 which **PROCGETMAZE** can get for you. **PROCGETMAZE** will give you a thirty by twenty maze, which fills Turtle's world. The finish point is where the letter **F** appears, and Turtle is already at the starting point. **PROCGETMAZE** also sets the paint to '.', so the trace of Turtle is not confused with the walls, which are given by a solid block.

Exercise

Write a program which makes Turtle find its way from the starting

point to the finish. You can only use Turtle with his pen down (he is not allowed to jump the walls!). Your program should also be capable of allowing Turtle to find his way around other similar mazes.

First maze

Work out a sequence of moves to get to the finish and edit your existing program accordingly. A possible program might be as follows:

```
>6000 DEF PROCMYTRY
>6010   PROCGETMAZE(1)
>6020   PROCFORWARD(3)
>6030   PROCRIGHT(90)
>6040   PROCFORWARD(20)
>6050   PROCLEFT(90)
>6060   PROCFORWARD(6)
>6070   ENDPROC
>RUN
>PROCMYTRY
```

This won't work, but it shows you the sort of format that is required.

Second maze

Once you have succeeded with maze 1, put maze 2 on the screen with a call to `PROCGETMAZE(2)`; work out a sequence of moves to solve it, write the corresponding program, and run it. If you don't succeed the first time, then keep trying.

Third maze

Put the third maze up on the screen with `PROCGETMAZE(3)`. Notice that it is different in that your moves are confined to a narrow channel (more like a conventional maze).

If you want, you can write a program consisting entirely of procedure calls just like the first two. However, there are more interesting things to be done. The maze program contains a number of functions which enable you to ask questions about Turtle's immediate environment. These, taken with `IF ... THEN ... ELSE` (if statements), `REPEAT ... UNTIL ...` (repeat statements) and `DEF PROC ... ENDPROC` (procedure

declarations) described earlier, will enable you to write much more powerful programs. These new functions are all described on the following pages.

Functions provided

FNFINISHED

This is used to see if Turtle has finished the maze; it returns the value **TRUE** if Turtle's current position is the same as the position of the finishing point. You would probably use it in a procedure of the following type:

```
DEF PROCSOLVEMAZE
  REPEAT
  :
  UNTIL FNFINISHED
ENDPROC
```

FNAHEAD

This gives Turtle a short, but useful sense of sight. It will give you the content of the next position toward which Turtle is heading. If this happens to be the edge of its world, then this character is that returned by **CHR\$(0)**. This is used as it is a **null character**—it does not print anything and does not have a special meaning to the computer, so it will never be used as a 'paint' and won't cause any odd things to happen.

This can be used to find out if the Turtle is by a wall by turning round and comparing the result of **FNAHEAD** with **CHR\$(240)** (the block character used is the 240th character in the set). You could use this to make Turtle turn to the right when it is blocked by a wall in front. If it isn't blocked, then it may as well move forward a step instead of turning. This could then be put inside a **REPEAT . . . UNTIL . . .** loop, like the one we've just used, to get the following short procedure:

```
>6100 DEF PROCSOLVEMAZE
>6110   REPEAT
>6120     IF FNAHEAD=CHR$(240) THEN
          PROCRIGHT(90) ELSE PROCFORWARD(1)
>6130   UNTIL FNFINISHED
>6140 ENDPROC
>RUN
>PROCGETMAZE(3)
```

Before trying out the new procedure, decide what you think is going to happen, then check with:

>PROCSOLVEMAZE

This shows some of the problems that a maze-solver has to deal with; the procedure above is obviously not clever enough to do the job. After the rest of the functions have been explained there are a few hints on how to solve these problems.

FNUNDER

This is similar to **FNAHEAD**, but it gives the character that Turtle is standing on at the moment. This function is not as useful as **FHAHEAD** when it comes to maze running, but you may find a use for it.

If you do use **FNUNDER**, remember that Turtle's paintbrush is attached to his tail, so the square is painted as he *leaves* it.

The next six functions all deal with positions in Turtle's world and so tend to be used together. In case you don't know, or have forgotten, a position *along* the screen is known as an 'X coordinate' and a position *up* the screen is known as a 'Y coordinate'.

FNXPOS and FNYPOS

These give the X and Y coordinates of Turtle's current position.

FNXFINISH and FNYFINISH

These give the X and Y coordinates of the finish of the maze. If you haven't called **PROCGETMAZE** yet, then these give a result that isn't in Turtle's world, namely $(-1, -1)$.

FNXSTART and FNYSTART

These give the X and Y coordinates of the start position of the maze and also refer to $(-1, -1)$ if you haven't asked for a maze yet.

To use these effectively, we must extend the **REPEAT . . . UNTIL . . .** loop we have just been using. The meaning of this statement is obvious in that you **REPEAT** some action **UNTIL** a condition becomes true. However, in this and the **IF . . . THEN . . . ELSE . . .** statement we have only used simple conditions so far, which aren't really powerful enough. For example, in **PROCSOLVEMAZE** above, Turtle won't cross a maze wall, but he might try to walk off the edge of his world, even though **FNAHEAD** could warn us of this. What we want to do is to see if a maze wall *or* a world edge is present, as in the new line following:

```

>6120 IF (FNAHEAD=CHR$(240)) OR
      (FNAHEAD=CHR$(0)) THEN PROCRIGHT(90)
      ELSE PROCFORWARD(1)

```

Note that we have put brackets around the simple conditions before joining them together. The result is also a condition, so that can be joined with another condition to make an even more powerful one and so on. It is also possible to join conditions by using **AND**, and a condition can be 'negated' by **NOT**, as in line 6140 following.

If your program makes Turtle return to the start, or you think it might, then you may decide that it isn't worth trying to solve that maze again. You could use something like this to stop this happening:

```

>6100 DEF PROCSOLVEMAZE
>6110   REPEAT
>6120     IF FNAHEAD=CHR$(240) THEN
          PROCRIGHT(90) ELSE PROCFORWARD(1)
>6130   UNTIL FNFINISHED OR
          ((FNXPOS=FNXSTART) AND (FNYPOS=
          FNYSTART))
>6140   IF NOT FNFINISHED THEN PRINT
          "I'VE FAILED!"
>6150 ENDPROC

```

Of course, this doesn't yet make any noticeable difference to the behaviour of Turtle, as **PROCSOLVEMAZE** still gets stuck without coming near to the finish or back to the start.

The trouble with this procedure is that Turtle makes no attempt to see why there is a wall in front of him; he just blindly turns to the right, and so misses the fact that there is a clear passage to his left. You have got to decide what sort of situations Turtle will come across in the maze, how to tell them apart and what to do in each case. So, you could start off by trying to alter **PROCSOLVEMAZE** so that Turtle can recognize the left-hand turns.

Third maze (continued)

Write a program to solve the third maze using these functions and constructs, and run it.

Fourth maze

How general do you think your last program is? If we were to give you another similar maze (i.e. one consisting of one narrow channel with a few twists and turns) would your program solve it even though you didn't know what the new maze looked like? Well, you can find out because the fourth maze is just like that. Take the maze 3 program, edit the call to `PROCGETMAZE` to access maze 4, and run your program without first looking at maze 4. If it works, fine! If not, can you see what assumptions you made that were specific to maze 3? Rewrite your program so that it handles both maze 3 and maze 4.

Fifth maze

Again, edit the call to `PROCGETMAZE` to access maze 5 and run your program without first looking at maze 5. This maze is like 3 and 4 except that it has T-junctions and blind alleys. Can you modify your program so that it works for this maze? (*Hint*: a program which always keeps contact with the wall on the left or right will work—a so-called 'wall-hugging' algorithm.)

There is also another way to use `FNAHEAD`, which you may like to use.

You can tell if you have already visited the next position by comparing the result of `FNAHEAD` with '.', because the paint trail Turtle leaves behind it in the maze consists of dots. You could use it to define a new function which will do this.

Defining a new function is very similar to defining a procedure; both start with `DEF`, but the function name must begin with `FN`. Also, we want our function to return a value. We have seen a value being assigned to a variable, such as `A=2+4`, where `A` is given the value 6. So, we might expect to put the function name, then an equals sign and finally the value. But, look at line 6120 above, where we have a function name, an equals sign and the value "", but which is supposed to be a call to that function.

To make sure that we have a distinctive way of giving the value, we simply don't put a name on the left-hand side of the equals sign. Also, when we do this it marks the end of the function, so we don't

have to use anything like `ENDPROC` in functions. So, we can finally define our function by:

```
>6200 DEF FNBEFORE
>6210 =(FNAHEAD=".")
```

There is no restriction on what goes on inside a function as compared to a procedure, so if you aren't sure that your new function is being used at the correct time, you could add:

```
>6205 PRINT "NOW USING FNBEFORE"
```

You don't need to use `FNBEFORE` to solve the fifth maze but you might like to experiment with it. This function provides some form of 'memory' of what you have done in the past.

A sample maze-solving program

If you are really stuck, here is a complete wall-hugging maze-solving program. Note that it won't work for mazes 1, 2, 6 or 7 (try it), as wall-hugging won't work on those sorts of designs.

```
>6000 DEF FN_wall
>6010 =((FNAH=CHR$(0)) OR
      (FNAH=CHR$(240)))
```

This stops Turtle hitting either a wall or the world's edge.

```
>6020 DEF FN_left_wall
>6030   LOCAL h%
>6040   PROCLT(90)
>6050   h%=FN_wall
>6060   PROCRT(90)
>6070   =h%
```

As the Turtle can only 'see' ahead of itself, to find out if there is a wall on the left it must turn to the left, see if there is a wall there, remember this by putting the result in the variable `h%`, turn to face forward again and finally pass on the result.

```
>6090 DEF FN_right_wall
>6100   LOCAL h%
>6110   PROCRT(90)
>6120   h%=FN_wall
```

```
>6130 PROCLT(90)
>6140 =h%
```

This does the same thing, but for a wall on the right-hand side.

We now have to decide what could happen when we hug a wall. If we are in a maze with channels, there are eight situations we can be in, which are shown in Figure 6.1. In each of these, Turtle has just moved upwards one square. Remember that Turtle can only ‘see’ one square away, so it can’t tell that (a) is a dead end until it moves right up to the end, as in situation (b).



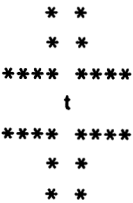
(a) In a channel (b) At a dead end (c) At a T-junction



(d) At a left turn (e) At a right turn



(f) At a left branch (g) At a right branch



(h) At a crossroads

Fig. 6.1 The eight possible situations that occur in a maze with channels.

The next procedure makes sure that in each of these situations Turtle will turn so that a wall is on his left-hand side, and then make a move along it.

```
>6150 DEF PROC_hug_left
>6160 IF NOT FN_left_wall
      THEN PROCRT(90):PROCFD(1):ENDPROC
```

This has dealt with situations (c), (d), (f) and (h) in Figure 6.1, where the wall has just turned left.

```
>6170 IF (FN_wall AND NOT FN_right_wall)
      THEN PROCRT(90):PROCFD(1):ENDPROC
```

This is for situation (e), where the corridor has bent to the right. Note that the order of these lines is important. For example, here we look for a wall in front and a space to the right and then do something if that has been found. Looking at the diagrams in Figure 6.1, this could occur in cases (c) and (e). However, we have already dealt with case (c), so we just need to take the action required here for (e).

```
>6180 IF NOT FN_wall
      THEN PROCFD(1):ENDPROC
```

In cases (a) and (g) we should just keep following that wall.

```
>6190 PROCRT(180):PROCFD(1)
```

Finally, we have only one situation left and so we don't have to make any more tests. In the dead end (b) we can only turn around and go back the way we came.

```
>6200 ENDPROC
```

Then, to solve the maze we just do this:

```
>6210 DEF PROC_TRY_HUG_LEFT
>6220   REPEAT
>6230     PROC_hug_left
>6240   UNTIL FNFIN
>6250 ENDPROC
```

The next two procedures apply the same ideas to following the wall on the right-hand side.


```

>6260 DEF PROC_hug_right
>6270   IF NOT FN_right_wall
        THEN PROCRT(90):PROCFD(1):ENDPROC
>6280   IF (FN_wall AND NOT FN_left_wall)
        THEN PROCLT(90):PROCFD(1):ENDPROC
>6290   IF NOT FN_wall
        THEN PROCFD(1):ENDPROC
>6300   PROCRT(180):PROCFD(1)
>6310 ENDPROC

>6320 DEF PROC_TRY_HUG_RIGHT
>6330   REPEAT
>6340     PROC_hug_right
>6350   UNTIL FNFIN
>6360 ENDPROC

```

Finally, this procedure allows us to try easily both methods on any of the mazes that we choose.

```

>6370 DEF PROCTRY(n%)
>6380   PROCGM(n%)
>6390   PROCCP('@')
>6400   PRINT "Left wall hugging."
>6410   PROC_TRY_HUG_LEFT
>6420   PROCJP(FNXS,FNYS)
>6430   PROCCP('%')
>6440   PRINT "Right wall hugging."
>6450   PROC_TRY_HUG_RIGHT
>6460 ENDPROC

```

Try this with mazes 3, 4 and 5 by typing `PROCTRY(3)` and so on.

Sixth maze

Wall-hugging won't work (try it). You're on your own now—good luck!

Seventh maze

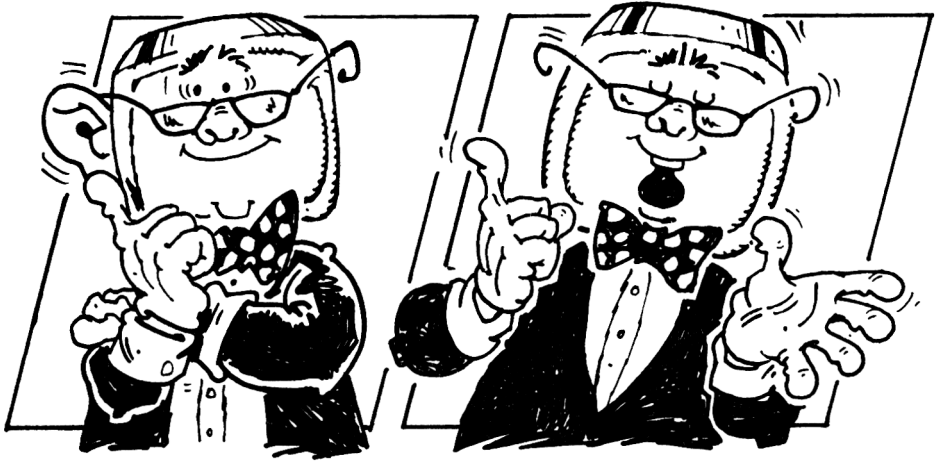
Here the constraints have been further removed—the arrow is no longer confined to a narrow channel. Don't worry if you don't get this far.

Turtle commands introduced

PROCGETMAZE PROCGM	Draws one of the seven mazes on the screen.
FNYFINISH FNYF	As FNYP0S , but for the finish position.
FNAHEAD FNAH	A function that gives the single character that lies immediately ahead of the Turtle. If the Turtle is at an edge of his world, this character is the null given by CHR\$(0) .
FNUNDER FNUN	A function that gives the single character that is underneath the Turtle.
FNFINISHED FNFIN	A function that can be used as a test to find out if the Turtle has reached the finish mark in the maze.
FNHEADING FNHE	A function that returns Turtle's current heading in degrees from North.
FNXP0S FNXP	A function that gives the current X coordinate of the Turtle.
FNYP0S FNYP	As FNXP0S , but gives the Y coordinate.
FNXSTART FNXS	As FNXP0S , but for the start position of the maze.
FNystart FNYS	As FNYP0S , but for the start position.
FNXFINISH FNXF	As FNXP0S , but for the finish position of the maze.

7

Input and Output



Input

In this chapter we will be looking at different ways of getting **information** into our programs.

We have seen a procedure to calculate an area of 5 units by 6 units earlier on in this book. Type in the following:

```
>NEW
>10 DEF PROCAREA
>20   LET L = 5
>30   LET W = 6
>40   LET AREA = L * W
>50   PRINT "AREA IS "; AREA
>60 ENDPROC
```

Now type:

```
>PROCAREA
```

You should get the following:

```
AREA IS 30
```

The procedure we have written only calculates an area of 5 units by 6 units, which is very limiting. It would be more interesting if we were able to calculate the area using several different lengths and widths, e.g. 5 units by 6 units, 8 units by 7 units, 12 units by 13 units. So let's change our procedure to do this.

Type in the following:

```
>NEW
>10 DEF PROCAREA
>20   FOR I = 1 TO 3
>30     READ L , W
>40     LET AREA = L * W
>50     PRINT "AREA is "; AREA
>60   NEXT I
>70   DATA 5,6,8,7,12,13
>80 ENDPROC
```

Now type:

```
>PROCAREA
```

You should get:

```
AREA IS 30
AREA IS 56
AREA IS 156
```

So what have we done? On line 30 we used a command called **READ**, which told the computer to read a piece of data and put it into a variable called **L**, and then read another piece of data and put it into the variable called **W**.

The computer finds the data on line 70 where we have a **DATA** command. The computer reads each pair of numbers from the data list (line 70) before moving on to the next pair.

Lines 30, 40 and 50 are all within our **FOR . . . NEXT** loop, which is repeated three times.

Now type in a different line 70 with a **DATA** command and six different numbers (do not forget the commas). Then type in:

```
>PROCAREA
```

You should have three different areas printed out.

So we can use a **DATA** command to input a sequence of data, and can get the computer to use this data with the **READ** command. It is possible to have several lines of data, such as the following:

```

>70 DATA 5,6
>72 DATA 8,7
>74 DATA 12,13

```

Before we go on to another form of input let us look at another BASIC command associated with **READ** and **DATA**: **RESTORE**.

Until now the computer, having read a piece of data, moves on to the next piece and so on until there is no more data. However, if the command **RESTORE** is used, the computer starts to read from the beginning of the data again. Try the following:

```

>75 RESTORE
>76 PROCAREA

```

This is an example of recursion (introduced in the first chapter) in which a procedure is called (i.e. line 76) within itself.

Now run your new version of **PROCAREA** by typing:

```
>PROCAREA
```

You will get:

```

AREA IS 30
AREA IS 56
AREA IS 156
AREA IS 30
AREA IS 56
AREA IS 156
AREA IS 30
:

```

Press **ESCAPE** to stop.

We have looked at two ways of getting data into the computer. The first was writing the data directly into the assignment statement, **LET**; the other way being to use the **READ** and **DATA** commands. However both these ways need the data to be written within the program. Having been written, it can only be changed by rewriting the particular line the data is in.

A third method allows us to actually input data while we are running the program. Type in the following

```

>NEW
>10 DEF PROCAREA
>20 INPUT "PLEASE ENTER LENGTH " L

```

```

>30 INPUT "PLEASE ENTER WIDTH " W
>40 AREA = L * W
>50 PRINT "AREA IS "; AREA
>60 PROCAREA
>70 ENDPROC

```

then

```
>PROCAREA
```

You should get:

```
PLEASE ENTER LENGTH
```

Type in a value for length and then press **RETURN**.

```
PLEASE ENTER WIDTH
```

Type in a value for width and then press **RETURN**.

```
AREA IS (your result)
```

```
PLEASE ENTER LENGTH
```

and so on. Having entered all your lengths and widths you use the **ESCAPE** key to finish.

This last method allows the user to enter any data, provided it is not nonsense. Also we can include a message, within quotation marks, which will be displayed to tell the user what sort of data the program expects to be entered.

To recap on the three methods we have used: the first was to input data directly, within a program statement. For example:

```
>20 LET L = 5
```

The second was to use the **READ** and **DATA** statements to allow several different pieces of data to be used. For example:

```

>30 READ L , W
>70 DATA 5,6,8,7,12,13

```

The last method was the command **INPUT** which allowed users of the program to enter whatever data they liked.

```
>20 INPUT "PLEASE ENTER LENGTH " L
```

Exercise

Now try to write two procedures, one called **PROCVOLUME1** and the other called **PROCVOLUME2**, both to calculate the volume of a cuboid (width $\times$ length $\times$ height). **PROCVOLUME1** should use the **READ** and **DATA** statements; **PROCVOLUME2** should use an **INPUT** statement.

Output

Let us now turn our attention to the outputting of information in BASIC. The main output command that we have already used is the command **PRINT**. The **PRINT** command allows the programmer to instruct the computer to display one of the following:

- (a) A piece of text in quotation marks. (**10 PRINT "FRED"**)
- (b) A number. (**10 PRINT 23**)
- (c) The contents of a variable. (**10 PRINT TEXT\$**
or **10 PRINT NUMBER**)

Try the following:

```
>NEW
>10 LET NUMBER = 25
>20 LET TEXT$ = "*BILL"
>30 PRINT "FRED",23,TEXT$,NUMBER
>40 END
```

Now type **RUN**. You will get:

```
FRED          23*BILL          25
```

The reason for the gaps between **FRED** and **23**, and ***BILL** and **25**, is that the screen is divided into sections which are 10 characters wide. If we number these 0 to 9, strings of characters (text) are started at position 0 of their section, and numbers are displayed with their last digit in position 9 of their section. The commas between **"FRED"**, **23**, **TEXT\$**, and **NUMBER** tell the computer to put the next output into the next available section. So we have:

```
0123456789012345678901234567890123456789 ← column
FRED          23*BILL          25      positions
```

Try typing line **30** with semicolons instead of commas.

```
>30 PRINT "FRED";23;TEXT$;NUMBER
```

Then RUN. You should get:

```
FRED23*BILL25
```

The semicolon instructs the computer to display the next character of the next output immediately after the last character it has displayed.

The comma can be regarded as an automatic form of the tab which you get on a typewriter. If you want to position your output in a particular position on a line, you can do this by the use of the function TAB. Try the following:

```
>NEW
>10 PRINT TAB(5) "FRED" TAB(15) "BILL"
>RUN
```

You will get:

```
0123456789012345678901234567890123456789 ← column
      FRED      BILL                      positions
```

The screen is divided into a number of columns, usually 0–39 (40 columns) and the value given to the TAB function is the number of the column where you wish to have the first character of the output. So TAB(5) "FRED" puts FRED on the screen starting at column 5 and TAB(15) "BILL" puts BILL on the screen starting at column 15.

The screen has a set number of lines, usually 0–24 (25 lines), so it would be nice if we could not only tell the computer what column we want our output to start in, but also what line to start on. Type in the following:

```
>NEW
>10 CLS
>20 PRINT TAB(17,1) "TOP"
>30 PRINT TAB(0,12) "LEFT"
>40 PRINT TAB(16,12) "MIDDLE"
>50 PRINT TAB(35,12) "RIGHT"
>60 PRINT TAB(16,24) "BOTTOM"
>70 END
>RUN
```

You should get the display shown in Figure 7.1.

We have given the TAB function two values. The first is the number of the column; the second is the number of the line. We can now put output to any part of the screen by the use of the TAB function. Try some outputs using the TAB function for yourself.

Before we leave our look at how to format output to the screen, there

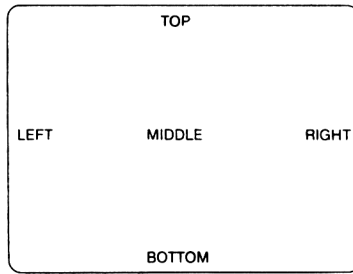


Fig. 7.1 The screen display produced by the TAB function.

is one more output function to examine. Often you may wish to leave a certain sized gap between different pieces of output on a line; perhaps a specific number of spaces between **FRED** and **BILL**. Try typing the following:

```
>NEW
>10 PRINT SPC(5) "FRED" SPC(15) "BILL"
>RUN
```

You will get:

```
FRED           BILL
```

Here the **SPC** function outputs a number of spaces to the screen; how many is determined by the value given to the function. The number in brackets is known as the **argument** of the function.

So, in the above, five spaces are output, then **FRED**, then fifteen spaces and then **BILL**.

To see the difference between **SPC** and **TAB**, type in the following:

```
>20 PRINT TAB(5) "FRED" TAB(15) "BILL"
>RUN
```

You should get:

```
FRED           BILL
FRED      BILL
```

In this chapter we have looked at ways of getting information into our programs and ways of outputting information. With practice, you will soon be very familiar with these methods.

Touch a key

Before we leave input and output, let us look at a function in BASIC

which allows us to use the keys on the keyboard as instructions to our program. Type in the following:

```
>NEW
>10 CLS
>20 REPEAT
>30   PRINT TAB(0,12) "PRESS THE
      SPACE BAR TO CLEAR THE SCREEN"
>40   LET HITKEY = GET
>50 UNTIL HITKEY = 32
>60 CLS
>70 END
```

Then type RUN. You should get:

```
PRESS SPACE BAR TO CLEAR SCREEN
```

The message will stay on the screen until you press the space bar.

What is happening is that the computer, after putting the message on the screen, waits for a key to be pressed. The code for that key is then placed in the variable HITKEY. This is what the function GET does: it returns the code value of any key that is pressed. Each key has a particular code number which is part of a sequence of numbers known as the ASCII code.

Lines 20 and 50 are in a special type of loop, a REPEAT . . . UNTIL loop. What happens is that lines 30 and 40 are repeated until the condition after the UNTIL command is satisfied—in this case, until the value returned by GET is 32 (the code number for the space bar). So the message remains on the screen until you press the space bar.

Exercise

Write a program that will put on the screen the messages shown in Table 7.1 when the keys indicated are pressed.

Table 7.1

Key	ASCII code	Message
L	76	LEFT
R	82	RIGHT
B	66	BOTTOM
T	84	TOP
M	77	MIDDLE

Each message should be put in the appropriate part of the screen to show its meaning. The program should continue to run until the key E is pressed, when it should finish.

The solution to this problem may be achieved in several ways. For instance, the program on the previous pages could be modified to include the appropriate message at line 30. The only snag is that we have to refer to the table each time we want to know the ASCII codes for the symbols L, R, B, T, and M. One way round this problem is to use the command:

```
>PRINT ASC("L")
```

The computer's response is the number which we are looking for.

Sometimes it is useful to know the converse of this routine; this is:

```
>PRINT CHR$(76)
```

```
L
```

```
>
```

In this way you can discover for yourself the appropriate codes for the above Exercise.

BASIC program commands introduced

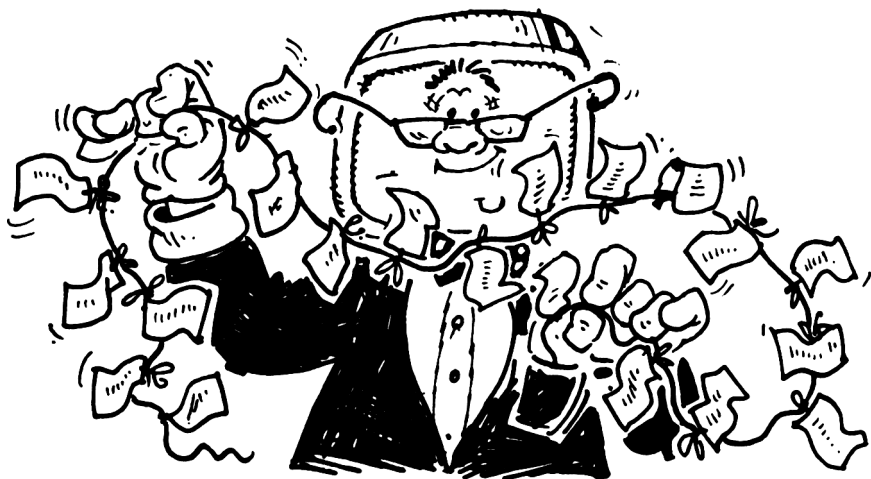
READ . . . DATA . . .	Reads the values from a DATA list into the variable names mentioned after the READ statement.
RESTORE	When multiple DATA lists occur, RESTORE enables the data pointer to be moved from one list to another.
INPUT	Inputs values from the user keyboard.
PRINT TAB(n)	Prints spaces up to a column position.
PRINT SPC(n)	Prints a number of spaces.

BASIC inbuilt functions introduced

ASC(character)	Returns the ASCII code value of the character.
CHR\$(number)	Returns the character corresponding to the ASCII code given by the number.
GET	Returns the code value of any key pressed.



Collections of Objects



Strings

Look back at the very first chapter of this book, when you were trying to make the computer do something (anything!) for you. If you try

```
>PRINT HELLO
```

and

```
>PRINT "HELLO"
```

the computer deals with each in an entirely different way. Do you know why this is? What do the quotation marks do? The answer was touched on in the *Ten Green Bottles* chapter, on page 19. It is very important that you should not think that the quotation marks (quotes) just contain words, or sentences, or quotations, i.e. that they mean pretty much the same as they do in a book.

They tell the computer not to attempt to interpret what is inside them; to treat them not as a list of instructions like the rest of the line, but simply as a list of characters. These lists we call **strings**, and they can be strings of letters, numbers, the odd characters around the keyboard, whatever you like; but until you instruct the computer, it will treat them just as lists of characters to be left alone. Let's put it another way:

putting quotes around anything will stop the computer from acting upon (evaluating) whatever it was you have put them around.

Let us try a few examples. When we tried

```
>PRINT HELLO
```

the computer first interpreted **PRINT** and got ready to be given something to print. When it read **HELLO**, because it did not recognize the word as belonging to **BASIC**, it assumed that the word must be the name of a variable, a label for a box somewhere, and it tried to find its value (evaluate it). Probably all it could find was that we hadn't given this label any particular value in the past, so it reported this failure to find anything.

Now try:

```
>LET HELLO=4
```

```
>PRINT HELLO
```

Now the computer has been given a value for a variable known as **HELLO**, and so it can find and print it when you give the computer those commands. Now try

```
PRINT "HELLO"
```

and you will see that, although the variable **HELLO** has a value, we have asked the computer to treat **"HELLO"** just as a simple list of characters, and to print it unchanged. Try

```
>PLEASE PRINT
```

and

```
>PRINT "PRINT"
```

We expect you can see that the command **PRINT** is followed by something either to evaluate and print, or to print directly.

Your computer is good at storing such lists (**strings**) as well as numbers, but we have to be careful to let it know what we intend. The **\$** sign after a variable name signals this to **BASIC**.

Let us combine strings of characters and numbers in a program. Try:

```
>NEW
```

```
>100 INPUT "What is your name ? " NAME$
```

```
>200 INPUT "What is your number ? " NUMBER
```

```
>300 PRINT
```

```
>400 PRINT NAME$ "'s number is ";NUMBER
```

```
>RUN
```

You should see:

```
What is your name ? HENRY
What is your number ? 682252
```

HENRY's number is 682252

Now let us assume we were to collect 10 names rather than one. We can write the following:

```
>100 INPUT NAME1$, NAME2$, NAME3$, NAME4$,
      NAME5$, NAME6$, NAME7$, NAME8$,
      NAME9$, NAME10$
>150 PRINT "The names are"
>200 PRINT NAME1$, NAME2$, NAME3$, NAME4$,
      NAME5$, NAME6$, NAME7$, NAME8$,
      NAME9$, NAME10$
```

The program clearly becomes much more tedious if, instead of collecting 10 names, we have to collect 100 names! The example shows that it is not always convenient to have a separate name for every variable used in a program.

Let us see how we might specify the action of the above program in mathematical notation:

```
input NAME$_i    for i = 1, 2, ... 10
print NAME$_i    for i = 1, 2, ... 10
```

A mathematician would immediately recognize that NAME\$ was the name, not of one variable, but of 10 variables, e.g.:

NAME\$_1, NAME\$_2, ..., NAME\$_10

In a programming language, we have to write statements all on one line and have therefore to adopt a different notation. In BASIC we write:

```
NAME$(I)    for NAME$_i
NAME$(1), NAME$(2), ...    for NAME$_1, NAME$_2, ...
```

Using this notation, we can rewrite the statement in the earlier tedious program as follows:

```
>100 LET N = 10 : DIM NAME$(N)
>200 FOR I = 1 to N : INPUT NAME$(I) :
      NEXT I
>300 PRINT "The names are"
```

```
>400 FOR I = 1 TO N : PRINT NAME$(I) :  
      NEXT I
```

Now in this version we have used the variable **NAME\$** which must be explained. **NAME\$** is the name of a group of variables which in BASIC is called an **array**. To declare such an array, we write

```
DIM NAME$(10)
```

where

DIM	is short for dimension
NAME\$	is the name of the group of variables
()	are brackets enclosing the range of the subscript

Let us add a few more lines to our program (between lines 200 and 300) to sort the names into alphabetical order before printing them.

```
>210 FOR I=1 TO 10-1  
>220 FOR K=I+1 TO 10  
>230 IF NAME$(I)>NAME$(K)  
      THEN R$=NAME$(I): NAME$(I)=NAME$(K):  
      NAME$(K)=R$  
>240 NEXT K  
>250 NEXT I
```

This is a fairly inefficient way of sorting objects, but it's an easy one to program!

```
>RUN
```

Exercise

Write a program which reads the names and telephone numbers of at least 20 people and prints an alphabetically ordered version of it (like a telephone directory). Produce an interactive program which:

- (a) greets the user by printing a message on the terminal and informs him/her of the services available;
- (b) enables the user to
 - (i) find somebody's telephone number,
 - (ii) find who has a telephone number,
 - (iii) add a new set of names/telephone numbers to the directory.

Arrays in two dimensions

Arrays of more than one dimension can be used in BASIC programs to represent groups of elements which need more than one subscript to identify them. We can store 25 letters from the alphabet in a 5×5 array using the program code:

```
>NEW
> 5 CLS
>10 DIM A$(5,5)
>20 RESTORE 1000
>30 FOR X = 1 TO 5
>40   FOR Y = 1 TO 5
>50     READ A$(X,Y)
>60   NEXT Y
>70 NEXT X

>1000 DATA A,B,C,D,E
>1010 DATA F,G,H,I,J
>1020 DATA K,L,M,N,O
>1030 DATA P,Q,R,S,T
>1040 DATA U,V,W,X,Y
```

The array may be visualized as shown in Figure 8.1.

Fig. 8.1 A 5×5 array for storing 25 letters of the alphabet.

	Y	1	2	3	4	5
X						
1		A	B	C	D	E
2		F	G	H	I	J
3		K	L	M	N	O
4		P	Q	R	S	T
5		U	V	W	X	Y

When the program is RUN, the DATA from line 1000 onwards is READ into the 5×5 array. We can easily test whether we have been successful or not by testing for elements at particular locations, e.g.

```
>PRINT A$(4,1)
P
```

and

```
>PRINT A$(2,5)
J
```

We will now add some further code to the program in order to display the data on the VDU.

```
> 90 CLS
>100 FOR X=1 TO 5
>110   FOR Y=1 TO 5
>120     PRINT TAB(X,Y); A$(Y,X)
>130   NEXT Y
>140 NEXT X
>150 PRINT
>160 END
```

Now run the revised program and note how the information is displayed.

Project

In earlier chapters you drew some pictures on the screen by calculating in advance the position of each character. This is a rather hard way of drawing pictures as it is fundamentally different from drawing by hand. People prefer to describe pictures in the form of movements rather than by calculations. A natural way of drawing can be achieved by using a Turtle graphics package like the one you met in Chapters 3 and 6.

Such a package introduces you to the world of the computer creature known as Turtle. Remember, this fictional creature can draw pictures for you on your TV screen. It is similar to drawing with a pen on a piece of paper, but it is Turtle who is holding the pen and you order Turtle rather than your hand to do the drawing. In this exercise you are asked to write a simple Turtle package of your own, from scratch.

The core of such a package is a two-dimensional PICTURE\$ array (Figure 8.2) which works as a temporary holding place for characters which will eventually be displayed line by line.

Fig. 8.2 The two-dimensional array which 'holds' characters for eventual screen display.

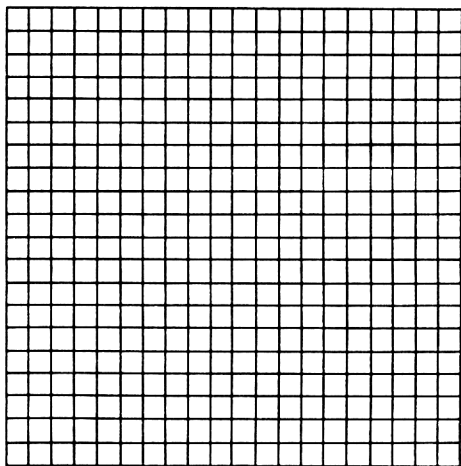
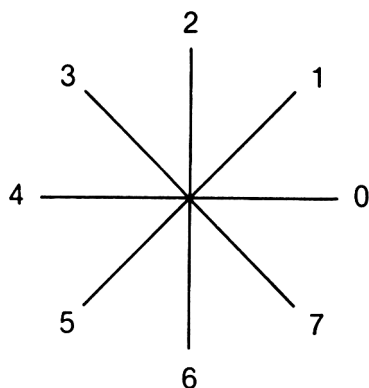


Fig. 8.3 The compass directions for moving Turtle.



The use of this array allows a **PROCDRAW** procedure to **PRINT** characters anywhere on the page, in whatever order makes life easier for the user. Without such a package, even drawing simple patterns would be very awkward to work out in advance for display in a line-by-line manner.

The system should first deposit blank characters in all the locations of the **PICTURE\$** array. The user then can deposit a character in a location of the array by using the **PROCDRAW** procedure.

You are asked to produce a simple set of procedures needed for a Turtle-like package. In addition to the **DRAW** procedure you will need at least one other procedure, to change the direction of drawing. The most simple set of directions is that of a compass (Figure 8.3). This means being able to say to which one of each of the adjacent locations of the array to move from the current location of Turtle.

You might consider having a **PROCTURNT0** procedure which points the Turtle to a specified direction. In addition, you will need a **PROCDISPLAY** procedure for showing the final array.

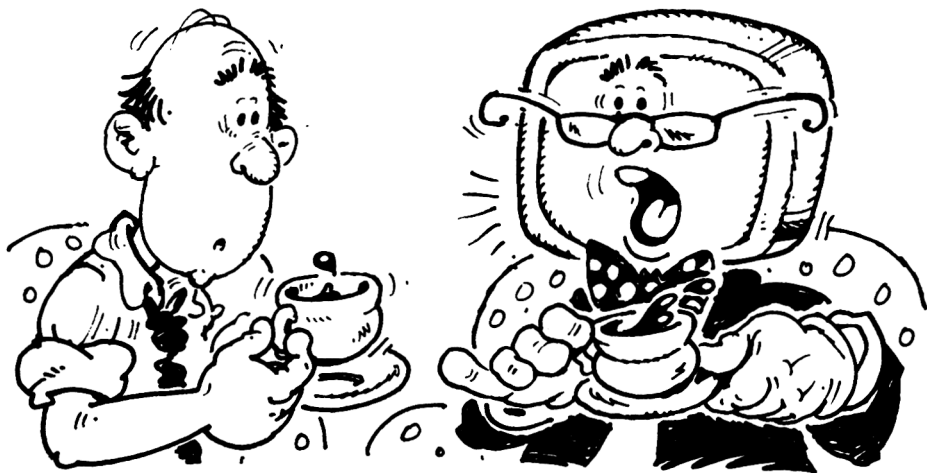
BASIC program command introduced

DIM name (. . .)

Defines an array.



Have a Chat with your Micro



More on strings

In the last chapter you learnt that although your computer is good at storing strings of characters of the alphabet as well as numbers, we have to be careful to let it know what we intend. In

```
>LET THING=4
```

LET tells the computer that we are going to put something in a named box, and it will assume, unless we tell it otherwise, that it will be a number. The value 4 is safely stored away with the label THING. (*Note: in the BASIC on your computer you don't need the LET; the computer will know from the rest of the line that you mean it. However, it is good to know about the LET, and to be able to use it, as some BASICs on other machines insist upon your putting it in.*) Now try:

```
>LET THING$="FOUR"
```

The dollar sign (\$) tells the computer to expect a **string** and not a value, so the quotation marks are important to confirm your intention. Try missing out the dollar, the quotes, or both, to see what happens; then you will know what has gone wrong the next time you forget them.

Remember, the messages that the computer prints when it cannot obey your command, or cannot decide what to do, are not there to tell you off! They are called reports, or error messages, or mishap messages and they often tell you exactly what the matter is, and help you put things right much faster.

Just as `THING` could have the value 4, or 40, or 45673982, so `THING$` can stand for "FOUR", or "FOUR GREEN BOTTLES HANGING ON THE WALL", or anything you like, up to certain length limits. Using `LET THING$=`, try making `THING$` stand for something really long, and then try:

```
>PRINT THING$
```

You can see that this will be very useful when we want some long statement printed out several times during a program, and there are many other important uses too.

The computer doesn't know anything about words or sentences until we have told it a lot more by writing procedures that help it to do this. To the computer, the space is just another character, and so "FOUR GREEN BOTTLES" is simply a string of 18 characters which includes two spaces. We have a very useful function in BASIC which tells us the length of a string. It is useful because we often don't know the exact length of a string which the computer has been interpreting for us. Try:

```
>PRINT LEN(THING$)
```

So `LEN()` will tell us the length of any string provided we put its name between the brackets. Now we really do know the answer to the old riddle! The things that a procedure or function has to act upon, or do something to, are called the **arguments** of that procedure or function; so `THING$` was the argument of the function `LEN()`. Now try

```
>LET THING$="FOUR GREEN"
```

and then

```
>PRINT LEN(THING$)
```

You will see that the computer has counted the space in the middle as a character. So " " is a perfectly good string, just as "" is. Yes, we *can* have a list with nothing in it, just as our variables can be 0 if we wish; and the empty list, or the empty string, is often very useful. Any mix of characters is permissible for a string, such as "HENRY IV, PART 2." or "AAAAAARRGGGHHHHH" or "Aaaaaaarrggghhhh!" or "***@#@TTTY\$", or " FOUR ", which the computer will see as different from "FOUR"; something of which you must be very careful when you are getting the

computer to compare strings for you.

Now here is something difficult, but important. Look at these, and try them.

```
>HELLO=4
>HELLO$="4"
>PRINT HELLO
```

followed by

```
>PRINT HELLO$
```

Is there any difference? Although the 4 looks the same both times, it does seem that the computer knows something different about each one, as it prints the number in a different place! There is a difference, of course, and an important one which will cause lots of problems in your programs until you know what is going on. We expect you realize that `HELLO` is the name of the place where the *value* 4 is kept, and `HELLO$` is the name of the place where the *character* 4 is kept. The important point is that you cannot mix values and lists of characters together: we could use `HELLO` in a sum, but not `"HELLO"` or `HELLO$`. `HELLO` holds a numerical value, but `"HELLO"` holds a list of characters, and `HELLO$` is the place-name of another list of characters (a one-item list).

```
>PRINT 2 + HELLO
```

is fine, as the computer looks up `HELLO` and finds 4 there, but

```
>PRINT 2 + HELLO$
```

confuses it immediately, as the `$` warns the computer *not* to evaluate what it finds but to leave it as characters. Try both of these. The error message is telling you that you are mixing up different types of things in a meaningless way. Similarly

```
>PRINT 2 + 2
```

is fine, but

```
>PRINT 2 + "2"
```

won't work.

Try

```
>LET THING$="HELLO"
```

Here `THING$` is the name we have decided to give to the place where we are keeping the string, and `"HELLO"` is what we have decided to put in

it. We can change either the name or the contents if we wish; so:

```
>LET THING$=""
>LET PONG$="HELLO"
```

Here we have reassigned `THING$`, and given "HELLO" to `PONG$` instead.

```
>LET PONG$="GOODBYE"
```

Here we have changed the contents of `PONG$`. If you want to be *really* awkward, try:

```
>LET HELLO$="GOODBYE"
```

and now

```
>PRINT "HELLO": PRINT HELLO$
```

Here, you see, is a way of proving that `BLACK$ = "WHITE"`, or even that `TRUTH$ = "LIES"`. I am sure you remember that the ':' in the line separates two commands, so that they can be written on the same line; again you cannot do this in every kind of BASIC, so you should learn to do things both ways.

There are a lot of things that you can do with strings when you have made them. You can check them against each other, to see if they match:

```
>IF HELLO$=PONG$ THEN PRINT "THEY MATCH"
  ELSE PRINT "THEY DON'T MATCH"
```

You can stick them together to make a longer and longer snake. Let's try some of these ideas in a program:

```
>10 THING$="HELP": MONSTER$=""
>20 REPEAT
>30   MONSTER$=MONSTER$+THING$
>40   PRINT MONSTER$
>50 UNTIL FALSE
>RUN
```

This will run out of space quite soon, and the computer will complain, but don't worry about that. Type `NEW` before trying any new ideas.

You can cut the strings into slices, and take pieces out or put them back at will. Try this new program:

```
>NEW
>10 LET MAGIC$="ABRACADABRA"
>20 FOR LOOP=1 TO LEN(MAGIC$)
```

```
>30 PRINT LEFT$(MAGIC$,LOOP)
>40 NEXT LOOP

>RUN
```

LEFT\$(X\$, Y) is a BASIC function which returns the left-hand Y characters of X\$. You can see if one list forms part of another list by adding a new line:

```
>35 IF INSTR(MAGIC$,"ACAD") THEN PRINT
    "IT FITS" ELSE PRINT "IT DOESN'T FIT"

>RUN
```

INSTR is a BASIC function which searches in the first string for an occurrence of the second string, returning the position of the substring.

All these different things are enormously useful if you want to write programs about words; to make the computer write poems; or (and this is what we are going to do now) to have conversations with your computer.

Conversing with a computer

Some of the problems in trying to have a conversation with your computer have a lot to do with the fact that it understands so few words, and can do so little with the words that it does know. So we have to write a program to help it respond to a few extra words, and to ignore the rest (not to evaluate them). After all, this is very like a human conversation, where most of the words flow over us and we respond only to a few key words. We have a very effective way of making the computer ignore everything we say, and not produce error messages by the thousand as it usually does if we try to be friendly. We can either put our remarks in quotes; or we can make the computer do it for us by using the INPUT statement and putting our remarks into a string variable like those in the last chapter, where they will be safe from the computer.

So, if we type

```
>HELLO, HOW ARE YOU?
```

the computer will have trouble, and tell us so. If we type

```
>"HELLO, HOW ARE YOU?"
```

the computer will have a different sort of trouble, because we have not

told it what to do with, or where to put the string we have entered. But here is a tiny program which represents the bare beginnings of conversational possibilities:

```
100 REPEAT
200   INPUT LINE CHAT$
300 UNTIL FALSE

>RUN
```

If you RUN this program, the computer will become the ideal listener; you can type anything you like and the computer will simply ask for more each time you press RETURN. It really doesn't much matter what the machine accepts or ignores in this program, as it never responds, and our conversation is very one-sided. At least we have stopped the error messages although the computer is throwing away everything you say, and overwriting each old comment with a new one! RUN the program, and then ESCAPE from it at some point and try .

```
>PRINT CHAT$
```

You will see what is happening. In a little while we will show you how to avoid losing your remarks if you want to keep them, and you will want to keep them when we have made the computer a better conversationalist. For the moment, we will improve our program by adding:

```
>290 PRINT "GO ON!"
```

Now try the program again; at least it makes a response.

```
>RUN
```

Before you sneer at this poor little program, just think how similar it is to many real conversations, where one person is the encourager and the other just runs on and on. The one thing that breaks up such a conversation is an inappropriate response, which betrays the fact that the other person wasn't listening, or didn't understand. We *know* the computer cannot understand, but we can easily make this harder to detect.

Clearly the computer must not be caught out by being asked a question and still be obliged to reply 'Go on!'. How do we detect questions in a conversation? By the rising tone of voice at the end of a sentence, which reminds us that we haven't been listening. In print, of course, we have the question mark, so if we can make the computer detect a question mark in our input it could make the appropriate response. We can do this easily by using the INSTR function (short for

INSTR(*string*) which looks for one string in another. If it finds one, then it returns a number which tells you at what position in the list of characters it was found, otherwise it returns 0, or logical **FALSE**. Press **ESCAPE** and then try:

```
>PRINT INSTR(CHAT$,"?")
```

If your last comment had a question mark in it, you will get its position in the comment printed out, otherwise just a 0. Now alter line 290 to:

```
>290 IF INSTR(CHAT$,"?") =FALSE THEN
      PRINT "GO ON!" ELSE PRINT "I'M NOT
      QUITE SURE. COULD YOU TELL ME A
      LITTLE MORE PLEASE?"
```

Now we have a very efficient converser going, and unless *you* forget the question mark, the computer will never get caught out. It is very easy to add to this program, so that the computer will respond differently to lots of different things. Suppose we wanted the computer to deflect all personal references in a modest sort of way, then all we would need to do would be to add something like this:

```
>280 IF INSTR(CHAT$," YOU ") THEN PRINT
      "NEVER MIND ABOUT ME"
```

As an important programming point, notice the spaces in the string " YOU ". If we did not have these spaces then *any* word which had **YOU** in it, such as **YOURS** or **YOUTH**, would cause the computer to respond—it really doesn't understand at all, and we have to help it a lot. Try inputting "HOW ARE YOU?" as a remark, and note the response that you get. You can now help the computer to respond to sentences ending in **YOU?** if you wish, but in any case, add a few more responses to our little program and you will see that it has a lot of possibilities. Always try out your additions several times, to make sure that peculiar things do not happen in your conversation—they probably will! Try a sentence that ends with **YOU** and see what happens.

Remembering what has been said

Our conversation is becoming quite a smooth one now, well worth letting someone else have a try with it to see if it goes wrong. However, there is one more important thing that we must show you, to make your conversation even better. Obviously, in a 'real' conversation, we expect

the person we are talking to to remember at least *some* of our remarks, and perhaps to refer to them occasionally—although it doesn't always seem to happen that way! Instead of merely overwriting one remark with another, the computer can be made to store up a list of all your remarks by using an array. Instead of just using `CHAT$` we will use `CHAT$(COMMENTNUMBER)`, and by making `COMMENTNUMBER` count up from one each time we make a remark, we will end up with a list of remarks called `CHAT$(1)`, `CHAT$(2)`, `CHAT$(3)` up to whatever `COMMENTNUMBER` has been reached. We need to:

- (a) alter all appearances of `CHAT$` to `CHAT$(COMMENTNUMBER)` and;
- (b) add a line at the beginning or the end of the `REPEAT` loop to set `COMMENTNUMBER` at a value one higher each time it goes round.

```
>110 COMMENTNUMBER=COMMENTNUMBER+1
```

Often you will have to think very carefully where such a line goes, as the result would be different in different cases, but in this case it will work in either position.

Before we `RUN` this, we must remember to warn the computer that we want it to make space available for such a list of strings by adding a `DIMENSION` line at the very beginning of the program.

```
>10 DIM CHAT$(20)
```

would do, if we only wanted to have twenty remarks in our list.

Now if you `RUN` this program, it will appear to behave in exactly the same way as before; you will not notice any difference, although the computer is storing up a list of your remarks inside itself. Let's make it show us what is going on. We will add another `REPEAT` command to make it print out the list so far each time, in a procedure that makes the computer count through its list until it reaches the current number, printing out each remark as it comes to it. Add this to your program:

```
500 DEF PROCPRINTCHAT
510   LOOP=0: PRINT
520   REPEAT
530     LOOP=LOOP+1
540     PRINT "COMMENTNUMBER"; LOOP
550     PRINT CHAT$(LOOP)
560   UNTIL LOOP=COMMENTNUMBER
570   PRINT
580 ENDPROC
```

Now you can use your program in the usual way, but whenever you want to see how the computer is storing your comments, press **(ESCAPE)** and call the procedure **PROCPRINTCHAT**.

How to be even friendlier

One way in which you can make a conversation seem real is by having the computer apparently learn your name, and use it in what can sometimes seem a very exasperating way. On the *Introductory Cassette* there is a short program called **GREETER** which uses a lot of these ideas, and the idea of arrays which we have just been talking about, to produce the effect of the computer 'recognizing' friends and learning about new people. See the first Exercise following for instructions for loading **GREETER**. You will also find this program listed in Appendix C. Such a little program can be used with scarcely any modification to make lists of telephone numbers or addresses for your use. The main addition would be to make the program store the information for you on cassette, by the use of the **PRINT#** command. For more information about how to do this, refer to your *User Guide*.

The way through the **GREETER** program is fairly simple; you **GREET** someone, you **CHECK** their greeting, you try to **IDENTIFY** them, and, if you fail to do this, you deal with them as a **NEWPERSON**. These are the names used for the procedures which do just this.

PROCGREET prints a greeting, and then looks in your response for the word **ILL**, and the word **NOT**, and the words **VERY WELL**. A suitable reply goes with each of these words, and **PROCHECK** is an all-purpose responder for anything you like to send. You can add a new response simply by adding another line (like lines 130–150) and inserting what you want the computer to look for, and what you want it to say if it finds it.

The variable **FOUND** simply records whether **PROCHECK** found what it was looking for or not. It can have the value 0 (**FALSE**) or -1 (**TRUE**). Look up **TRUE** in your *User Guide* if you need to. By line 160, we need to have something to say if all our searches have failed, i.e. if **FOUND** is still **FALSE**.

PROCIDENTIFY asks for your name to see if it needs to add you to its list, and we need to say a little about this list. It is very like the one in our little conversation program, except that there are *two* lists kept, one of names and one of remarks. These two arrays keep in step with each other, so that if your name is in **NAMES(3)**, then the remark that applied to you would be in **REMARKS(3)**. You can see what an effective way this

is of keeping sets of information tidy and related to each other. By counting through one list looking for, say, a name, we can be sure when we find it that the remark, or telephone number, or address, is in the same position on the other list. If **PROCIDENTIFY** finds you, then it prints out the corresponding remark; otherwise, it hands you on to **PROCNEWPERSON** where a new name slot is waiting for you.

Exercise

Load the **GREETER** program from your *Introductory Cassette* and try it out. With the tape positioned as it was after loading **SOLVER**, turn the cassette to side A (do NOT rewind). Type

>CHAIN "GREETER" and press **RETURN**

and then press the **PLAY** button on the recorder (**CHAIN** combines **LOAD** and **RUN**). **STOP THE TAPE IMMEDIATELY** once **GREETER** has loaded (the computer will beep to indicate this).

When you're bored with the program's reponse as it stands, modify **GREETER** so that it responds to a different set of remarks. Alter the program so that it asks for useful information. By using **PRINT#** (look this up in the *User Guide*) see if you can make the computer store some information for you.

Exercise

Alter the **TEN GREEN BOTTLES** program so that instead of printing things like **3 Green Bottles** it prints **three green bottles**. Much better! You can do this with an array, just like the ones we have been using, thus:

```
DIM NUMBER$(10)
NUMBER$(1)="ONE"
NUMBER$(2)="TWO"
:
```

Or, if you are knowledgeable about these things, you could use

```
FOR LOOP= 1 TO 10
  READ NUMBER$(LOOP)
NEXT LOOP
```

and

```
DATA ONE,TWO,THREE,FOUR,FIVE,SIX,  
SEVEN,EIGHT,NINE,TEN
```

Then, all you have to do is to alter the print commands from

```
PRINT I "GREEN BOTTLES"
```

to

```
PRINT NUMBER$(I) "GREEN BOTTLES"
```

BASIC inbuilt functions introduced

INSTR(string,string)
for 'in string'

Searches the first string for an occurrence of the second string, returning the position of the substring.

LEFT\$(string,number)
for 'left string'

Returns the left-hand number of characters in the string.

LEN(string)
for 'length'

Returns the length of the given string.

RIGHT\$(string,number)
for 'right string'

Returns the right-hand number of characters in the string.

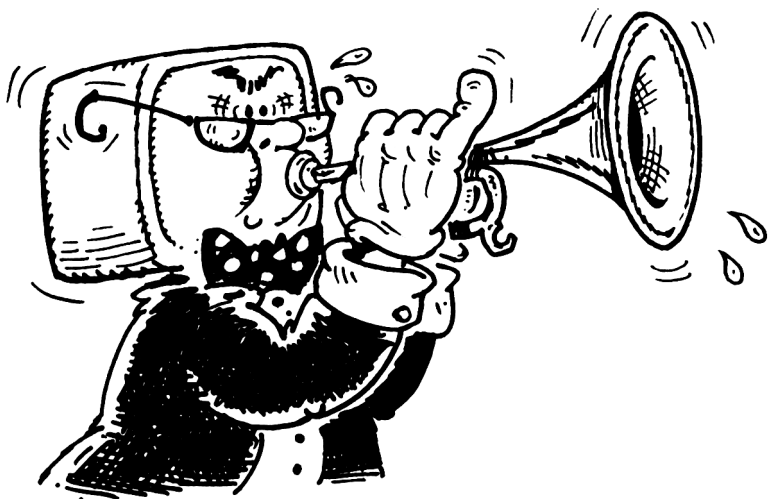
BASIC system command introduced

CHAIN

Combines **LOAD** and **RUN**.



Sounding out your Electron



Introduction

The Electron microcomputer has on-board circuits which are capable of producing sounds using appropriate commands via a small internal speaker. Naturally the quality of the output from so small a speaker is not brilliant, but it should be possible to further amplify the signals produced in order to drive a speaker of higher quality. For the moment let us explore some of the machine's capabilities.

Making sounds with your microcomputer

The microcomputer has the ability to produce up to four separate channels, or **voices**. A keyword is needed in order to specify a particular sound, and this keyword must be followed by four numbers, separated by commas, which will tell you the type of sound you will hear. For example, if you type in the command

> SOUND 1,-1,53,100 and press **RETURN**

you hear a pure tone whose **pitch** corresponds to a note of middle C (frequency = 262 Hz) on a piano keyboard, and which lasts for five

seconds; whereas the command

> SOUND 2,-1,89,50 followed by **RETURN**

produces a sound whose pitch corresponds to A (frequency = 440 Hz) above middle C on the piano keyboard, and which lasts for two and a half seconds.

In general, the SOUND command will take the form

SOUND C,E,P,D

where C corresponds to a channel number, 0 to 3;

E corresponds to an envelope number;

P corresponds to the pitch, 0 to 255;

D corresponds to the duration of the note, 1 to 255.

With the Electron microcomputer it is possible to produce only one sound at a given time. The length of time for which the tone will last is selected by varying the final parameter in the SOUND statement, D, which is set in twentieths of a second and can take any value from 1 twentieth to 255 twentieths. The corresponding sound then lasts for between 0.05 and 12.75 seconds.

In the two simple examples above, you set the envelope parameter to -1, and in so doing told the computer to produce a pure tone or note with no pitch or frequency change during the time for which the note was sounding.

If you sit in front of a piano keyboard and play the middle C, which is roughly in the middle of the keyboard, followed by the next seven white notes, D, E, F, G, A, B and C, you will have played a scale of C major. Then, if you were really clever and repeated the exercise starting with middle C and progressing in semitones (i.e. including the black keys as well), you would have played a chromatic scale of C major, having twelve semitones. This corresponds to an octave; the frequency of the last note is double that of the starting note, which had a frequency of 262 Hz. This is all very well, but we are trying to learn to play our microcomputer, not become a concert pianist!

Let us write a short program to instruct the microcomputer to play the scale for us. Try this one for yourself:

```
>10 REM *** A chromatic scale ***
>20 INPUT "WHAT IS YOUR STARTING NOTE?"
    START
```

```

>30 FOR N=START TO (START+48) STEP 4
>40   SOUND 1,-1,N,10
>50 NEXT N
>60 END

```

Test-run the program, trying starting notes of 53, 69, and 89; hopefully you will have played perfect chromatic scales of C major, E major, and A major respectively, just with one finger press! Much simpler than playing the piano, isn't it?

Exercises

- 1 Modify your program so that it plays a descending scale instead of an ascending one. Can you make it play longer notes?
- 2 Now further modify the program so that it plays an ascending scale, followed, after a brief pause, by a descending one.

Controlling the sounds produced by your microcomputer

We can now incorporate the **ENVELOPE** parameter into the sound statement by setting the value of **E**, the second number in the sound statement, to a value other than -1 or 0 . This will have the effect of causing the pitch to vary while the note is sounding. It is necessary to define the envelope by means of an **ENVELOPE** statement before invoking the associated **SOUND** command in which it is to be incorporated.

In order to define an envelope we use a command of the form:

```

ENVELOPE N,SL,P1,P2,P3,SN1,SN2,SN3,
126,0,0,-126,126,126

```

You will see that this statement involves no less than fourteen numbers. Each of the first eight numbers determines some aspect of the envelope shape which is being defined. They provide information which the computer needs to have in order to instruct it how to vary the pitch whilst the note is sounding. The need to define so many numbers makes the use of the envelope statement seem unnecessarily complicated until we look more carefully at the construction of the statement.

The envelope is made up of three distinct parts, as is clear from Figure 10.1. The slope of the graph for each section needs to be either positive

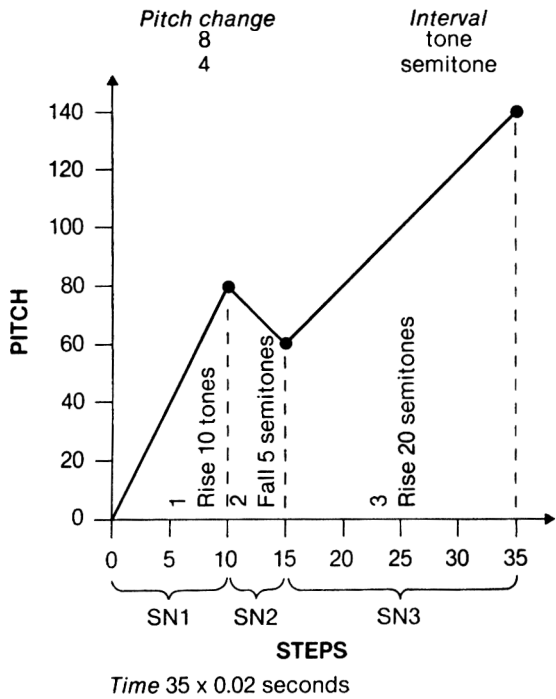


Fig. 10.1 The parts of the envelope.

(uphill) or negative (downhill), and is determined by the value we choose as the **pitch increment** (i.e. P1, P2, or P3) which is made positive or negative as appropriate. We can also choose how many steps there are to be in each of the three sections, by means of the **step number** (i.e.. SN1, SN2, or SN3). Finally we need to state the length of the time step associated with each of the pitch increments mentioned above.

Table 10.1 shows how this works in practice.

Table 10.1

Parameter	Value	Note
SL	2	step length 2*1/100 seconds
P1	+8	go up in steps of 1 tone
P2	-4	go down in steps of 1 semitone
P3	+4	go up in steps of 1 semitone
SN1	10	10 steps in section 1
SN2	5	5 steps in section 2
SN3	20	20 steps in section 3

In all there are thirty-five steps in one complete envelope cycle, and each of these lasts for 0.02 seconds. It follows that one complete envelope lasts for 35×0.02 or 0.7 seconds. Now, if we set the sound statement to be of duration greater than 0.7 seconds, the sound will repeat for a number of times, owing to the **auto-repeat** facility. Here is an example for you to try, based on the information above:

```
>30 ENVELOPE 1,2,8,-4,4,10,5,20,126,0,0,
    -126,126,126
>40 SOUND 1,1,53,140
```

In this short program you will see that the sound lasts for $140 \times 1/20$ seconds, but that the envelope is designed to last for 0.7 seconds; therefore the event will repeat itself ten times. Now we can add some extra program code to our short program above, as follows:

```
>10 FOR I=1 TO 5
>20   PITCH=RND(255)
>40   SOUND 1,1,PITCH,140 : REM *** this will
    replace the old line 40 ***
>60 NEXT I
```

Can you anticipate what will happen when you run the complete program?

The complete program should read as follows:

```
>10 FOR I=1 TO 5
>20   PITCH=RND(255)
>30   ENVELOPE 1,2,8,-4,4,10,5,20,126,0,0,
    -126,126,126
>40   SOUND 1,1,PITCH,140
>50   REM *** One SOUND command gives
    10 repetitions of event ***
>60 NEXT I
>70 END
```

Exercises

- 1 Design a program using the **SOUND** command, incorporating envelope control, which will produce a variety of interesting sounds from your microcomputer. Remember that you may define at least three different sounds in your program, each having its own associated envelope control; but that only one sound may be produced at any time.

- 2 Explore the possibility of inputting the data for a sound program by means of the READ and DATA statements.

How to turn your micro into a simple musical instrument

It is quite easy for us to turn the microcomputer keyboard into a simulated piano or organ-type keyboard. After all, the micro works with numbers, which are input to the machine as we press the individual keys. You will recall that in an earlier chapter we told you how to find the ASCII codes which corresponded to particular keys, using the CHR\$ function. Furthermore we know that our microcomputer can produce sounds whose pitch depends on yet another set of numbers, e.g. a value of the pitch parameter PITCH = 89 corresponds to a note of frequency 440 Hz. We will use this knowledge, together with our previous programming experience, to design a simple musical instrument. Don't type these lines yet—just try to understand them. The full program listing is given later on.

Firstly we identify which keys on the micro keyboard we wish to use as the 'piano' keys.

```
110 LET KEY$="AZSXC FVGBNJKM, L. / : "
```

is a string of characters to represent the keys.

```
430 LET PITCH=INSTR(KEY$,CURRENTNOTE$)*4+33
```

defines the pitch of the notes used in the program, calculated from a low value of PITCH=33 which corresponds to a note of G.

```
440 SOUND 1,-1,PITCH,255
```

chooses the type of sound we want to make. Later you can experiment with this for yourself by including an envelope parameter in this expression. Notice that we have given the sound a very long duration: we will have to interrupt it if another key is pressed.

When getting a new key from the keyboard, we will include a check to make sure that it is among the keys included in KEY\$ at line 110.

```
330 DEF PROCGETNOTE
340 REPEAT
350   LET NTE$=INKEY$(2)
360   UNTIL NTE$<>CURRENTNOTE$ AND
      INSTR(KEY$,NTE$)
370 ENDPROC
```

There is another problem that must be overcome: the auto-repeat facility must be speeded up when the program is run, so that each note is sounded continuously while the key is depressed. But it must be reset before the program ends, or you won't be able to type anything! (Try it!) This is done as follows.

```
120 *FX 11,1
130 *FX 12,1
```

to speed up the auto-repeat;

```
200 *FX 12,0
```

to reset it to normal.

This introduces another problem. If we (ESCAPE) from the program, or if any other error occurs, line 200 will not be encountered, and the auto-repeat will still be too fast. To overcome this we introduce an **error-trapping** command, which will tell us what the error was and where it occurred, and then reset the auto-repeat before ending the program.

```
500 ON ERROR REPORT: PRINT" AT LINE ";
    ERL: GOTO 200
```

Finally it would be nice if the program were self-documenting; text should be produced on your screen which explains to some extent what is happening. This is accomplished by means of the procedure called **PROCINSTRUCT**, starting at line 230 in the full program which is listed following.

```
>100 REM PIANO PROGRAM, starting with initializations
>110 LET KEY$="AZSXCFCVGBNJKM,L./: "
>120 *FX 11,1
>130 *FX 12,1
>140 PROCINSTRUCT
>150 LET CURRENTNOTES$=""
>160 REPEAT
>170   PROCGETNOTE
>180   PROCPLAYNOTE
>190 UNTIL CURRENTNOTES$=" "
>200 *FX12,0
>210 END
>220
>230 DEF PROCINSTRUCT
>240   CLS
```

```

>250 PRINT""YOUR PIANO KEYBOARD"
>260 PRINT""The appropriate keys are:""
>270 PRINT"    A S    F G    J K L    : "
>280 PRINT"        Z X C V B N M , . / "
>290 PRINT""(The letter C represents middle C.)"
>300 PRINT""Play on...press space to stop."
>310 ENDPROC
>320
>330 DEF PROCGETNOTE
>340 REPEAT
>350 LET NTE$=INKEY$(2)
>360 UNTIL NTE$<>CURRENTNOTES$ AND INSTR(KEY$,NTE$)
>370 ENDPROC
>380
>390 DEF PROCPLAYNOTE
>400 SOUND &11,0,0,0: REM To interrupt
      the presently playing note.
>410 LET CURRENTNOTES$=NTE$
>420 IF NTE$="" OR NTE$=" " THEN ENDPROC
>430 LET PITCH=INSTR(KEY$,CURRENTNOTES$)*4+33
>440 SOUND 1,-1,PITCH,255
>450 ENDPROC
>500 ON ERROR REPORT: PRINT" AT LINE ";
      ERL: GOTO 200

```

When you have completed the task of typing in the program code, you can run your program. You will need to use a little imagination in order to visualize the piano keyboard, but with a little practice you could become quite proficient at your new musical instrument.

Why not try to modify the program for yourself, so as to incorporate one or more envelope statements which could be called up as the program is running? In this way you could devise several different sounds which can be played with the 'piano'.

Sound commands introduced

SOUND channel, envelope,
pitch, duration

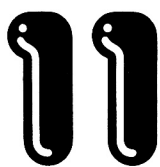
Generates sounds with the
given parameters.

ENVELOPE N,SL,P1,P2,P3,
SN1,SN2,SN3,126,0,0,
-126,126,126

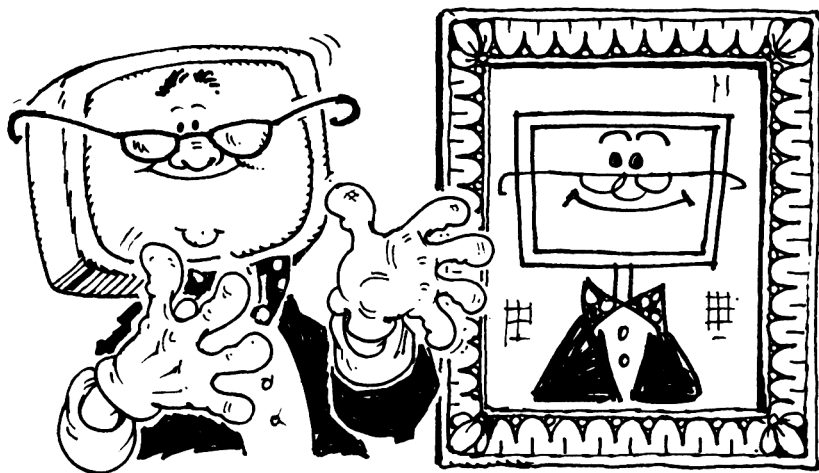
Controls the pitch variation
of the sound generated.

BASIC system and language commands introduced

ERL	Gives the line number where the error occurred.
GOTO <i>n</i>	Makes the computer follow the instructions on line numbered <i>n</i> .
INKEY (<i>n</i>)	Waits for a specified (<i>n</i> hundredth of a second) to see if a key is pressed, and then returns the character pressed.
ON ERROR	When the computer detects an error it will follow the instruction following this command.
:	Separates two BASIC statements written on the same line.
*FX <i>n, y</i>	Deals with the running of the computer, as specified in the <i>User Guide</i> .



Pretty Pictures



Introduction

The Electron has a very powerful set of commands to allow you to draw pictures on the TV screen, in a variety of colours. There are a number of methods that you can use to give different effects. The number of ways of combining these is so large that we can't explore them all here, so you must do a lot of experimenting with the commands. Be sure to keep notes on what you do; one of the biggest problems with this sort of graphics is that sometimes the 'mistakes' come out better than the effect you were aiming for!

Telling the machine to use graphics

To allow you to choose whether you want to use the greater part of the memory to hold your program, or to use a better graphics display, there are a number of screen **modes**. These give different numbers of points displayed on the screen, different sizes of text, and different numbers of colours displayable at a time. To change mode, you use the **MODE** command, followed by a number from 0 to 6. The machine starts off in mode 6, so typing

>MODE 6

will always give you the standard display.

The flicker when you change modes is quite normal, and you should note that a mode change clears the screen and resets any changes to the format that you may have made. So changing mode will always put you back to where you started, just in case you make a mistake.

The different modes available are:

MODE 6

This gives 25 lines each of 40 characters of text, allowing two colours on the screen at any time—see below for a note about this. It doesn't allow the use of graphics.

MODE 5

This suddenly gives very chunky text, only 20 characters per line, but with 32 lines on the screen, which you might find useful when showing something to a group of people. It gives you graphics with 160 points across the screen and 256 from top to bottom. Text and graphics can use up to four colours on the screen at a time. This mode uses half the memory available on the Electron.

MODE 4

Here you have 40 characters on 32 lines, and graphics of 320 points across by 256 points up the screen. But you are back to two colours at a time. This mode uses the same amount of memory as mode 5.

MODE 3

This allows 80 characters per line on 25 lines, in two colours, but with no graphics. This mode uses over half the memory available on the Electron.

MODE 2

This is like mode 5, but it allow up to sixteen colours at a time on the screen, which is the maximum possible with this computer. This mode uses about two-thirds of the memory available on the Electron.

MODE 1

This is like mode 4, but with up to four colours allowed. It uses the same amount of memory as mode 2.

MODE 0

This allows 80 characters on 32 lines and gives the smoothest graphics display, but only allows two colours to be used at a time. It uses the same amount of memory as modes 1 and 2.

Note: The methods required to use all the available colours in each mode will be explained later. However, for now you should know that when it says 'two colours' this means the *background* is one colour, normally black, and the *text and graphics* are another, normally white. There are ways, which there isn't room to go into here, of using any of the computer's colours for these two; so you could have, say, red writing on a blue background, or even text flashing from yellow to blue on a background which flashes from red to green—not something we would really recommend. For these details, try reading the *User Guide* when you have finished this chapter.

Try out all of these modes. To show the effects of the different **resolutions** of the display in each mode (i.e. the number of points available) use the following lines in each of them:

```
>MOVE 0,0
>DRAW 1000,900
```

This will draw a slanted line on the display (except in the text-only modes). If you don't have enough memory, because you already have a large program in memory, you will get the message:

Bad MODE

There will also be a line number if this happens while you are running a program. This means that the computer will not allow you to damage your program by using the memory for another purpose. For similar reasons, due to the inner workings of Electron BASIC, you can *not* change mode within a procedure or function. Any attempt to do so produces the same error message.

Because it gives good definition, most of the examples will use mode 4, but you can of course try them in other modes to see the effect. To make life a little easier, here is a short program that you should use:

```
>1 REM Windows Program for Graphics
>2 MODE 4
>3 MOVE 0,196:DRAW 1279,196
>4 VDU 28,0,31,39,26
>5 VDU 24,0;204;1279;1023;
```

```
>6 VDU 29,0;204;
>7 MOVE 0,0
>8 END
```

The workings of this program won't be explained here, but you must be careful to get the punctuation correct when you type it in.

The program divides the mode 4 screen into two areas, with a line between them—there is actually a small gap on both sides of this line. The lower area gives you six lines of text, so that you can call the procedures to draw into the graphics area in the upper section. Using this allows you to be sure that your typing and graphics can't overlap, which is very useful when you are experimenting like this. The graphics area appears as though it has been shoved upwards, so all your procedures will work properly. However, this does of course mean that you have a little less height available in the graphics area than if you simply used mode 4.

To clear the graphics area on its own, just type the BASIC command which stands for 'CLear Graphics':

```
>CLG
```

To clear the text area, use the 'CLear Screen' command:

```
>CLS
```

The easiest way to clear the whole screen is to type

```
>RUN
```

again.

How the screen is organized

Because each mode gives you a different number of points on the screen, Electron BASIC uses a simple method to make sure that your programs give similar results no matter what mode you use. It believes that the screen, when in a graphics mode, always has 1280 points across the screen and 1024 up the screen. So if you tell the computer to draw a line that will go, say, halfway across your screen, it will always pick the right number of **pixels** to light up. A **pixel** is the name given to the actual points in the display, to distinguish them from the large number of points that the computer *thinks* it has; the name is a bad abbreviation for 'PICTure ELEment'. This does mean that sometimes you can draw two lines that are close together, but only see one line on the screen.

This is because each pixel is used to represent a group of points that are close together. This effect shouldn't worry you too much, but to demonstrate it try typing the following:

```
>MODE 0
>MOVE 1000,0
>DRAW 1000,1000
>MOVE 1007,0
>DRAW 1007,1000
```

Try this in other modes.

Drawing lines

We have just been using two simple statements to draw lines.

```
MOVE x,y
DRAW x,y
```

As mentioned above, these refer to points and not pixels, so each takes two variables, an x-coordinate and a y-coordinate. The xvalues range between 0 and 1279, and the y values range between 0 and 1023. These values can, of course, be produced by an expression, a function call, or by using variables.

When you use the **WINDOWS** program, as noted above, the y value has a maximum value of 819.

The use of these statements requires the idea of a **graphics cursor**. This is a pointer to the place on the screen which we have last visited. This graphics cursor starts at position (0,0), which is at the bottom left-hand corner of the screen, after the **MODE** command.

The **MOVE** statement moves the graphics cursor to the stated location; the **DRAW** statement moves the graphics cursor to the stated location, drawing a straight line as it goes. The following procedure will draw a square on the screen:

```
>100 DEF PROC SQUARE
>110 REM draw a square of fixed size in a
      fixed place.
>120 MOVE 300,300
>130 DRAW 600,300
>140 DRAW 600,600
>150 DRAW 300,600
>160 DRAW 300,300
```

```
>170 ENDPROC
>RUN
>PROCSQUARE
```

Note that the **MODE** commands have to be *outside* the procedure definition. You will recall that the **MODE 4** was in the **WINDOWS** program, which is still part of the program we now have in memory, and when you **RUN** will be invoked. However, if you do not have the **WINDOWS** program you need a **MODE 4** command before you call **PROCSQUARE**.

The program works by moving the graphics cursor to position (300,300) (i.e. 300 steps in the x direction and 300 steps in the y direction). The cursor then moves to (600,300), drawing a straight horizontal line 300 steps long. The program then draws a vertical line 300 steps long to (600,600), and then draws the remaining two lines of the square.

Exercise

Modify the procedure so that it draws a triangle inside the square with only its corners touching the sides of the square. Also, modify it so that you can say where you want the square to be drawn and how big it should be, by giving the procedure three parameters.

Drawing solid triangles

In order to draw solid objects it is usually only necessary to construct them from triangles, e.g. a solid square can be made from two solid triangles, as shown in Figure 11.1. It would therefore be nice to be able to draw solid triangles. The Electron computer has this ability, which requires the use of the **PLOT** statement. The **PLOT** statement needs three values, i.e.:

```
PLOT what,x,y
```

The **what** part controls exactly what is to be plotted. The **x** and **y** values are again the **x** and **y** coordinates, to say where the graphics cursor is to go. We want to draw triangles, so the **what** section must be set to **85**:

```
>MODE 5
>PLOT 85,500,500
```

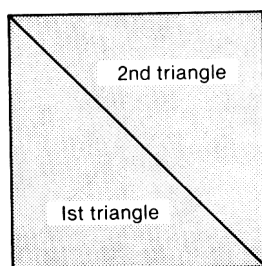


Fig. 11.1 A solid square can be made up from two solid triangles.

Notice that all that appears is a diagonal line from the bottom left-hand corner of the screen. This is because we need three points to give the three corners of the triangle. The `PLOT` statement gives the computer one corner and the other two are taken from the last two points that the graphics cursor has been sent to. Thus, to draw a solid triangle in the middle of the screen, we need to write a procedure as follows:

```
>MODE 4
>200 DEF PROCSOLID_TRIANGLE
>210 REM Draws a single solid triangle
>220 MOVE 200,200
>230 MOVE 1000,200
>240 PLOT 85,600,600
>250 ENDPROC
>RUN
>PROCSOLID_TRIANGLE
```

To draw a square on the screen we need to draw two triangles. The first draws the bottom left-hand half of the square; the second draws the rest of the square, i.e. the top right-hand half. Thus the points are visited as shown in Figure 11.2 and a procedure to draw a solid square is as follows:

```
>MODE 4
>300 DEF PROCSOLID_SQUARE
>310 REM Draw a single square of fixed size
      and position
>320 MOVE 200,200
>330 MOVE 200,600
>340 PLOT 85,600,200
>350 PLOT 85,600,600
>360 ENDPROC
>RUN
>PROCSOLID_SQUARE
```

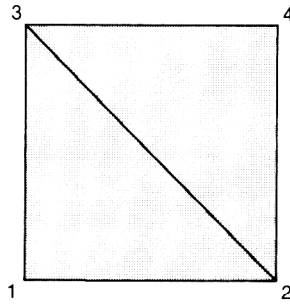


Fig. 11.2 The order in which the points are visited to draw two triangles which form a solid square.

Drawing a man

The first stage in drawing any picture on the computer is to forget about the computer for a while and sit back with a pencil to design the object on a piece of paper, using only straight lines. So, for example, in drawing a man, we note that he has two legs, two arms, a body and a head (Figure 11.3).

Next, decide on a scale for the various parts so that your picture looks rather like a draughtsman's design. Remember to put the position of each point on your diagram. Then divide the picture into triangles (assuming you want a solid object). The result might look something like Figure 11.4.

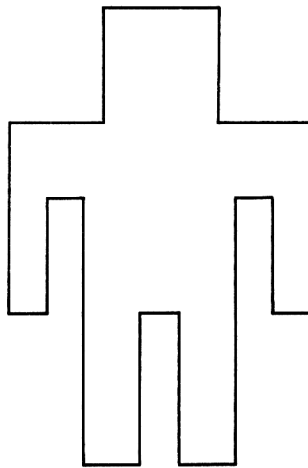


Fig. 11.3 Rough design of a man, using only straight lines.

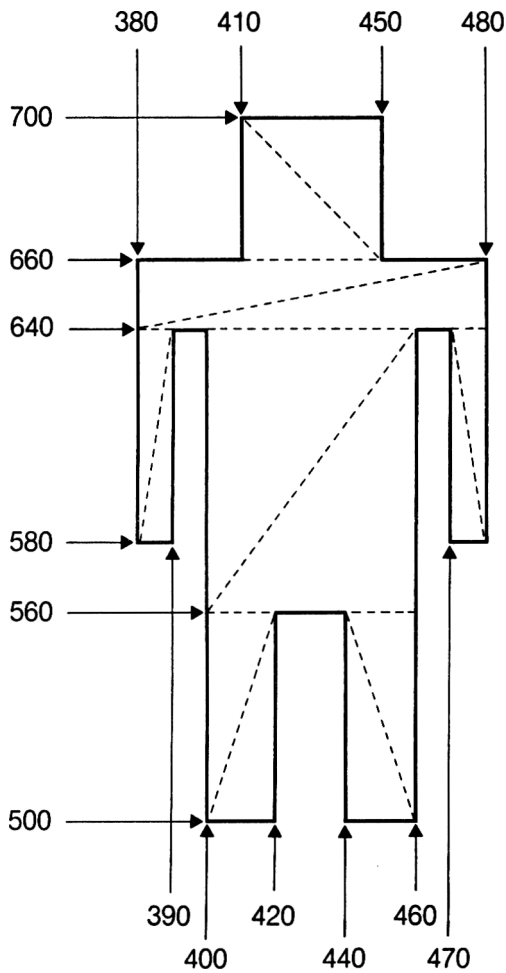


Fig. 11.4 Your scale drawing of a man might look like this. Note how it is divided into triangles.

Now start the procedure to draw your figure,

```
>MODE 4
>400 DEF PROCDRAWMAN
>410 REM Draw a man
>420 MOVE 400,500
>430 MOVE 420,500
>440 PLOT 85,400,560
>450 PLOT 85,420,560
>460 REM Left leg drawn
```

```
>470 MOVE 440,500
>480 MOVE 460,500
>490 PLOT 85,440,560
>500 PLOT 85,460,560
>510 REM Right leg drawn

>520 REM NOTE we are already at one
        corner of
>530 REM one of the triangles in the body,
>540 REM so there is no need for two MOVES
        here.
>550 MOVE 400,560
>560 PLOT 85,460,640
>570 PLOT 85,400,640
>580 REM Body drawn

>590 MOVE 380,650
>600 MOVE 380,640
>610 PLOT 85,480,650
>620 PLOT 85,480,640
>630 REM Shoulders drawn

>640 MOVE 480,640
>650 MOVE 470,640
>660 PLOT 85,480,580
>670 PLOT 85,470,580
>680 REM Right arm drawn

>690 MOVE 390,640
>700 MOVE 380,640
>710 PLOT 85,390,580
>720 PLOT 85,380,580
>730 REM Left arm drawn

>740 MOVE 410,650
>750 MOVE 410,690
>760 PLOT 85,450,650
>770 PLOT 85,450,690
>780 REM Head drawn

>790 REM Finished the whole man
>800 ENDPROC

>RUN
>PROCDRAWMAN
```


Try running this program in mode 5 and then in mode 2. You may notice that the man appears to have his hand in his pocket. Now try mode 0 and then mode 1 to run **PROC**DRAWMAN. The problem here is the one that you were told about above: two lines which should appear separately being drawn in the same place.

Try altering this procedure to have it draw a woman.

Other **PL**OT options

The **PL**OT statement is actually capable of a lot more things than we have shown here. In fact, **MOVE** and **DRAW** are just shorthand forms of **PL**OT commands, to make the most common activities easier to perform. The complete set of values for the 'what' part of the **PL**OT command is very large and may get larger as new versions of the computer are made. So the list here is just of the most useful ones at this stage:

PLOT 4,x,y — does the same as **MOVE** x,y

PLOT 5,x,y — does the same as **DRAW** x,y

PLOT 7,x,y — draws line in background colour (often black) to x,y

PLOT 13,x,y — draws a dotted straight line to x,y

PLOT 15,x,y — draws a dotted line in black to x,y

PLOT 69,x,y — puts a point at x,y

PLOT 71,x,y — puts a black dot at x,y

PLOT 85,x,y — draws a triangle to x,y

PLOT 87,x,y — draws a black triangle to x,y

Note that after a **PL**OT statement the graphics cursor is always at x,y.

There is another set of useful values, which deal with **relative coordinates**. This means that the x value specifies how far up the graphics cursor should move from wherever it is, and the y value says how far sideways it should move. The **PL**OT commands which do this are:

PLOT 0,x,y — Move to the point x,y relative to the current position.

PLOT 1,x,y — Draw a line to x,y relative to the current position.

PL0T 3,x,y — Draw a line in background colour (often black) to the point **x,y** relative to the current position.

PL0T 17,x,y — As for 1, but with a dotted line.

PL0T 19,x,y — As for 3, but with a dotted line.

PL0T 65,x,y — Plot a single point relative to the current position.

PL0T 67,x,y — As for 65, but in black.

PL0T 81,x,y — Plot a triangle in white, taking the third point relative to the current position.

PL0T 83,x,y — As for 81, but in black.

These are a lot of numbers to try to remember, but if you compare the lists you will see a simple relationship between the relative and absolute versions of the commands, and even between those that do similar things but in opposite colours.

To show you how useful relative movements can be, the following procedure will draw a square on the screen, but it will be placed wherever the graphics cursor happens to be at the time:

```
>MODE 4
>1000 DEF PROCREL_SQUARE
>1010 REM Draw a square of fixed size, but
      anywhere
>1020 PL0T 1,300,0
>1030 PL0T 1,0,300
>1040 PL0T 1,-300,0
>1050 PL0T 1,0,-300
>1060 ENDPROC
```

Now, try this in various places

```
>MODE 5
>RUN
>MOVE 100,100
>PROCREL_SQUARE
>MOVE 600,200
>PROCREL_SQUARE
```

and so on.

For something completely different, try to write a procedure to draw a solid house with two windows. You can then add a door, chimney and so on. (*Hint*: you may find it easier to draw a house without windows

and then put in the windows using `PLOT 87,x,y`).

Do this with absolute `PLOT`s, then convert to relative. Finally, you can draw a whole street by calling this procedure a few times. This will start you on the way to drawing complex pictures, and shows how easy it can be. But it also shows that you have to decide when the picture is complete; for example, you could still add some fences between houses, use different shapes and sizes of house, and so on. But before you try this, finish this chapter—otherwise you will never get round to it.

Adding colours

Mode 5, as explained earlier, allows four colours on the screen at once; until now we have only seen two of these colours on the screen at a time. The colours are controlled by the `GCOL` command, which requires two variables, e.g.:

`GCOL operations, colour`

For this chapter we will ignore the operations value and say it should always be zero. The colour value depends on what colour you want. In mode 5 these are:

- 0 — black
- 1 — red
- 2 — yellow
- 3 — white

Thus

```
>GCOL 0,1
```

sets the colour to red, and any further `DRAW` or `PLOT` commands that use 'white' in the tables above will now draw in red until another `GCOL` is used. So, using the last example, to give our man a red jumper we just add the lines:

```
>545 GCOL 0,1
>735 GCOL 0,3
>MODE 5
```

Note that we use `MODE 5` now to see the colour effect.

```
>PROCDRAWMAN
```

Assuming that you drew your woman with a dress, make it yellow.

Drawing circles

Drawing circles can be done by the following procedure, which takes three arguments: the x -coordinate of the centre, the y -coordinate of the centre, and the radius of the circle (see Figure 11.5). After the circle is drawn, the graphics cursor is left at the centre of the circle.

```
>MODE 4
>1200 DEF PROCCIRCLE(X,Y,R)
>1210   LOCAL I,J
>1220   FOR I=Y+R TO Y-R STEP -4
>1230     J=SQR(ABS(R*R-(I-Y)*(I-Y)))
>1240     MOVE X-J,I
>1250     DRAW X+J,I
>1260   NEXT
>1270   MOVE X,Y
>1280 ENDPROC
```

There is not enough space in the chapter to explain how it works, but the example below will demonstrate its use. The man of the previous example is now given a round head, centred on (430,670) with a radius of 10 units.

```
>740 PROCCIRCLE(430,670,20)
>DELETE 750,770
>MODE 1
>PROCDRAWMAN
```

Give your woman a round head.

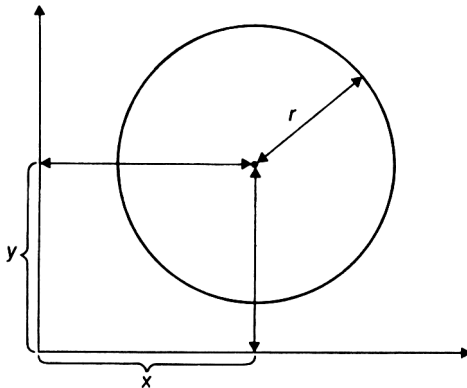


Fig. 11.5 The coordinates for drawing a circle.

A way of making things easier

One way to make programming of the drawing of objects easier is to store the PLOT numbers in DATA statements and READ them in. One problem is the lack of PLOT numbers to change colour or to plot circles. This problem can be overcome by making up PLOT numbers for these operations and then checking for those numbers when reading the data. For example, PLOT 100,x,y might mean 'change colour to colour x' while PLOT 102,a,b followed by PLOT 103,c,d might mean PROC-CIRCLE(a,b,c). This last possibility can be difficult to program in; the following example of drawing a man *assumes* that PLOT 102,a,b is followed by a PLOT 103,c,d.

```

>MODE 4
>2100 DEF PROCDRAWMAN
>2110 LOCAL I,P,X,Y,A,B
>2120 REM Draw a man using data statements
>2130 FOR I=0 TO 26
>2140 READ P,X,Y
>2150 IF P=100 THEN GCOL 0,X
>2160 IF P=102 THEN A=X:B=Y
>2170 IF P=103 THEN PROCCIRCLE(A,B,X)
>2180 IF P<100 THEN PLOT P,X,Y
>2190 NEXT I
>2200 DATA 4,400,500,4,420,500,85,400,560,
85,420,560
>2210 DATA 4,440,500,4,460,500,85,440,560,
85,460,560
>2220 DATA 100,1,0,4,400,560,85,460,640,85,
400,640
>2230 DATA 4,380,650,4,380,640,85,480,650,
85,480,640
>2240 DATA 4,480,640,4,470,640,85,480,580,
85,470,580
>2250 DATA 4,390,640,4,380,640,85,390,580,
85,380,580
>2260 DATA 100,3,0,102,430,670,103,20,0
>2270 ENDPROC

>MODE 5
>PROCDRAWMAN

```

Exercise

Try drawing the following objects:

a fox;
a chicken;
a bag of grain;
a boat;
and a river.

We shall need them later!

BASIC commands introduced

MODE . . .	Changes mode to give different sizes of characters and pixels, using different amounts of memory.
MOVE . . ., . . .	Moves the graphics cursor.
DRAW . . ., . . .	Draws a line to the given position.
PLOT . . ., . . ., . . .	Draws various lines or shapes.
GCOL 0, . . .	Changes colour.
CLS	Clears the text area.
CLG	Clears the graphics area.

(These two screen areas normally overlap, so then there is no difference in what the last two commands do.)

12

Computer Games



Introduction

We are going to talk about a very old puzzle now, and show you how it can be implemented on a computer. Most games, and not only fantasy games, take place in their own little world. *Draughts (Checkers)* takes place in a very small one, for example: a board of 64 squares, inhabited by black and white pieces. *Chess* takes place in a world a little more varied: same terrain, but the inhabitants are a little more exciting. *Noughts and crosses (tic-tac-toe)* has a very boring, limited world, while something like *Dungeons and Dragons* has a very varied and interesting one. We are used to an extremely complicated world, with far more happening in it than we can control or notice at any one time. You could say that when we play games we are making little models of bits (or subsets) of our world, so that we can look at them more closely and control them better—we are much more the God of the chessboard than we are of the world around us!

This idea has interested people for a long time, and a lot of writers have written books about made-up worlds of their own. Probably one of the most carefully worked out is *Through the Looking Glass* by Lewis Carroll, which you probably know. The author actually makes use of a lot of the characteristics of the chess world in his story, and keeps to quite a lot of the rules of the game.

Every world has two aspects. One aspect is what is *in* the world: what its inhabitants are like, what the world itself is like. The other is about the *rules* that run the world. The chess-world rules are well enough known; every sort of inhabitant has his own way of living. Our own world has millions of rules—so many that we can't know them all, and things often look chaotic because we don't know by what rules they are governed. In general, things fall toward the centre of the earth, water freezes at around 0 degrees Celsius, human beings see with their eyes, and so on; and we learn to expect that most things will behave in their own way.

The little world that we are going to look at is called *Riverworld*, because it is about a river. First then, what is in this world? Not much: only a river, with banks on either side, and a boat floating on it; a man, a fox, a chicken, and a sack of grain. The situation is that the man is going to market, and has to get over the river with his animals and grain. It's a very simple world. But we haven't come to the *rules* of this world yet, and they are what make things difficult.

If the fox is left alone with the chicken, he will eat it. If the chicken is left alone with the grain, she will eat it. The boat will only hold one thing, plus the man to row it.

Question: How does the man get himself and all his things to the other side?

Try to solve the puzzle on paper if you don't know the solution already. Write down each stage of the solution.

Turning the game into a program

We have already said that all games are models of parts of our own world; they use bits of familiar things to build up small units of worlds with rules. Similarly, the computer program must be a model of the game itself. The computer has to use the things *it* knows to build up something like the game; and we have already come across the idea that procedures can describe or model separate actions. In this game, for example, if we move the chicken into the boat, we take the chicken away from wherever it is, and add it to whatever is in the boat. So obviously, something like

```
PROCTAKEOUT("CHICKEN", "WHEREVER")
```

and

```
PROCPUTIN("CHICKEN", "BOAT")
```


will be involved, and you will see later that this is so. Perhaps we could make up a bigger, general procedure called **PROCMOVE**, which does both for us; something like:

```
DEF PROCMOVE(WHAT$,WHERE$)
  PROCTAKEOUT(WHAT$, WHEREVER$)
  PROCPUTIN(WHAT$,WHERE$)
ENDPROC
```

This, too, is very much what does emerge.

Also, somewhere in the program there has to be a bit that 'sets up' the world, and models the objects in it, and where they are (in the program this is **PROCSTARTOFF**). Elsewhere is a bit that makes sure the player keeps to the rules of the world (this is **PROCILLEGALMOVE**). The program needs a lot of 'housekeeping' bits as well, to keep the model in its current 'state' as we play the game.

Appendix D contains a listing of the **RIVERGAME** program. It has some pretty rudimentary graphics which we hope you can understand after the last chapter. If you do not have the *Electron Introductory Cassette*, you can type the programs from the listings and save them on a blank cassette.

Load the **RIVERGAME** program with the *Introductory Cassette* tape positioned as it was after loading **GREETER**. Type

```
>CHAIN "RIVERGAME" (RETURN)
```

and press the **PLAY** button on the recorder.

Now play with the program a little, taking a look at the program's listing (in Appendix D) to see if you can understand what is happening. The program contains a few things that were left in to show you how it is working; you may like to remove these when you use the program simply as a game.

When you **RUN** the program, you will see a page of instruction on the screen, with a procedure name. As you go on with the program, the procedures called will also list themselves on the screen. This is to show you exactly what the program is doing between stops, and it is a very valuable way of helping to debug any program in its earlier stages. All we did was to write

```
PRINT "PROC[NAME]"
```

or

```
PRINT "FN[NAME]"
```

after the **DEF PROC[NAME]** or **DEF FN[NAME]** lines of the program. You

can of course remove these lines later if you wish; or better, you can *edit* in a **REM** just after each of the numbers of those particular lines, which will 'disable' them, and yet allow you to get them back if you want them again. To get them back again, just remove the word **REM**.

If you have not gone on from the instructions, now press any key. At the top of the screen you will see a list of procedures; in the middle is a very simple picture of the setup; and below this you are being asked for your instructions for moving grain, fox, chicken, man and boat.

The rules of this gameworld are built into the program, and you will not be allowed to get away with much!

Try moving the man (**M**) into the boat—we hope you remember that this is unwise!—and amongst the function and procedure names, you should see a warning message. You can also see exactly which procedure has produced the warning. Although the picture shows the Riverscene after the accident, you can see a **PROCSTARTOFF** towards the bottom, which shows you that the game has been abandoned and restarted. Now go ahead and try to get everything on the other side of the river. You can leave the program with (**ESCAPE**) at any time, or you can retrace your steps in the game. The messages will tell you if anything is not allowed (from **PROCILLEGALMOVE**) or goes wrong (from **PROCDISASTROUSMOVE**).

Going a bit deeper

Here are the various components of our program, with rather more detail of what they do; look at them with the listing of the program.

After the introductory page of instructions, **PROCSTARTOFF** sets everything up for a new game. All the things and places in the drama are represented by letters in the computer. They are kept in two lists (arrays) called **POSITION\$(0,N)** and **POSITION\$(1,N)**, and each of these lists contains three empty slots. **POSITION\$(0,N)** contains the three places in our world, so **POSITION\$(0,1) = "LB"** (Left Bank), **POSITION\$(0,2) = "B"** (Boat), and **POSITION\$(0,3) = "RB"** (Right Bank). **POSITION\$(1,N)** contains all the objects (**G,F,C,M,B**) that are in these places. **POSITION\$(1,1)**, for instance, will contain the objects that are at **POSITION\$(0,1)**.

At the beginning of a game all the objects are on the left bank, so in **POSITION\$(1,1)** are the initials of all the objects, corresponding to **POSITION\$(0,1)** which holds the label **"LB"** for left bank; the functions **FNTAKEOUT** and **FNPUTIN** keep the lists up to date as things move

around. You can stop the program at any time with **(ESCAPE)**, and have a look in the **POSITION\$(1,N)** list. You will find that the objects have kept pace with the progress of the game. "N" is always 1, 2, or 3, corresponding to the left bank, the boat, and the right bank. Every time **PROCDISASTROUSMOVE** reports positively, the game is abandoned and **PROCSTARTOFF** is called to set the game up again.

PROCWHATNOW begins each move, so it calls **PROCSHOWIT** which draws a little picture of the current position, and then asks for the next move. Then it calls **PROCMOVE**, which does most of the substantial work of the program.

PROCMOVE(WHAT\$,WHERE\$) first ensures that the rules are not being broken, by calling **PROCILLEGALMOVE**. If they are, the move is abandoned, and we go back to **PROCWHATNOW**. If all is well, then **FNTAKEOUT** and **FNPUTIN** do the 'housekeeping' and adjust the **POSITION\$(1,N)** list to keep track of the game. **PROCDISASTROUSMOVE** is now called to see if anyone has been eaten or drowned; it finds out who has been eaten by calling **FNSUPPER**. If anyone has gone, the game is aborted, and is restarted via **PROCSTARTOFF**.

If no disaster has occurred, **PROCBOATPOS** makes sure that the boat is on the correct side of the river, with its occupants. This little function is necessary as the **LB** and **RB** do not move about, but the **B** does!

When the move has been completed, **PROCWHATNOW** is called for the next move, which begins by a display of the current position via **PROCSHOWIT**.

There are two other procedures which we have not mentioned yet; one of them is a 'housekeeping' program called **FNTRANSLATE**. See if you can work out what it does—it's not madly exciting. The other we shall talk about in the next section.

Programming the rules

There is no doubt that the two most complicated looking procedures are **PROCILLEGALMOVE** and **PROCDISASTROUSMOVE**. This is because they are doing so many things, which we must admit is a *bad* thing. **PROCILLEGALMOVE** already calls **PROCALREADYTHERE**, which takes care of one of the illegal moves; and it would be easy to make each rule have its own little procedure, so that all **PROCILLEGALMOVE** did was to call them one by one, like this:

```
DEF PROCILLEGALMOVE
  PROCALREADYTHERE
```

```

PROCNOBOAT
PROCNOMAN
PROCHELPLESSBOAT
:
ENDPROC

```

This would improve the readability of the program enormously, and produce a very much better organized (elegant) program altogether.

If you want to try writing games for your computer yourself, it would be a good idea to work out a few of the rules in this game, and then see how they are written into the program. Here are a couple for you.

If you try to move something to where it already is, this is obviously a useless move. How does the computer check on this? Think about `PROCMOVE(WHAT$,WHERE$)`. If the computer finds `WHAT$` in `POSITION$(1,2)` then it knows that `WHAT$` is at whatever is in `POSITION$(0,2)`; and so if `POSITION$(0,2) = WHERE$` then obviously (we hope!) `WHAT$` is already there.

If you want to move an object from one place to another, the man must always be there to do it. So, if we are trying to move the chicken somewhere, then we must first find where the chicken is, and then check that the man is in the same place as the chicken. See if you can find the bit that does this—you know it will be in `PROCILLEGALMOVE`.

The boat cannot move without the man either, so whenever the object to be moved is "B", the program must first check that . . . what?

Try to sort out some of the rest of the rules—notice how much easier this would be if you had rearranged `PROCILLEGALMOVE` as we suggested! How does the program know if you have solved the puzzle, by the way? (It really is terribly crude!)

Learning to live without 'sugar'

The `RIVERGAME` program really works at two levels. One of them plays the game, looks after the rules, and moves the objects about; the other shows you the pictures, asks you for your next move, does all the kind, good-mannered things that make life easier. It's just a sweetener, really; sugar sprinkled over the serious stuff.

What we are getting at is that you don't *have* to do it the way `PROCWHATNOW` does it. You can call, from the keyboard, any procedure or function that you wish. If you `(ESCAPE)` from the program in the middle of the game and call `PROCSHOWIT`, then you will find that

you are the master really. You could do the next move by calling `PROCMOVE("C","B")`, shall we say, and you can re-enter the normal way through the program by saying `GOTO` (whatever line number the program escaped at).

In fact, you can try all the procedures and functions in the game separately to see what happens. Don't forget to give them some *arguments* if they need them (those are the things in the brackets after the procedure or function name). We expect that you will get some pretty weird results from some of them. Just one word about the difference between a function and a procedure—in simple language, the difference is that a function actually ends up having the value of some result that it has worked out, almost as if it were some kind of variable; while a procedure simply works through the lines inside it. So, if you want to see what result your function is producing, you must ask the computer to print it out; for instance:

```
PRINT FNTHING
```

or

```
PRINT FNTAKEOUT("M", POSITION$(1,1))
```

In fact, you can play the whole game from start to finish by calling procedures from the keyboard. Alter `PROCSTARTOFF` so that it does not call `PROCWHATNOW` (put a `REM` at the beginning of that line). Now call `PROCSTARTOFF`, and then, by using `PROCMOVE("C","B")` to move the chicken into the boat (or whatever move you please) play the whole game through just by calling `PROCMOVE(...)` as many times as you need to.

You will notice that many of the procedures and functions in this program call each other; in fact they are joined in a kind of chain. Another way (and a very good way) to have written it would have been to make them all quite independent of each other, and to have a procedure which did nothing but call them all in the right order.

Exercise

Use the above technique to write a procedure that will make the whole game play itself right through! It could start:

```
DEF PROCSOLVEIT
  PROCSTARTOFF
  PROCMOVE("C","B")
```

```

    PROCMOVE(you fill it in)
    PROCMOVE(you fill it in)
    :
ENDPROC

```

Extraordinary idea! You just type **PROCSOLVEIT**, and the whole thing rattles itself off automatically. You can put a few **PROCSHOWITs** in it if you want to see what is happening, or just put one at the end.

How to write your own game

You will, no doubt, be writing your own games and it is worth dividing up your program in the way we have been suggesting. Ask yourself these questions:

- (a) How am I going to represent the Gameworld?
- (b) How am I going to represent the objects or people moving around in it?
- (c) What are the rules for moving?
- (d) What are the rules for stopping illegal moves?
- (e) What are the rules for detecting winning or losing moves?
- (f) What information has to be provided during the game?
- (g) Which part of the program plays the game, and which part just looks pretty, or makes it easier to play?

You can answer these questions by playing the game through, and noting down the procedure for each stage or each move or each rule. Give each of these stages a name, and then they can be your procedure names when you come to the programming.

Noughts and Crosses is quite a good one to start with; so is a *Magic Square Maker*; but whatever you choose, be prepared to be *very* patient! Designing a longish program is a long job, and debugging it often a longer one—that's why knowing your error messages is so important.

Exercise

Appendix D presents a version of the **RIVERGAME** referred to in this chapter, which does not have any fancy graphics.

However, as part of the exercise of Chapter 11, you should have

designed the necessary objects to make this game more interesting. Alter the program so that the game played by it remains unchanged but add the good graphics.

The **RIVERGAME** program on your introductory cassette is *one* solution to this exercise but not a very good one. Why is this so?

Reflections

After reading this book we hope you have learned the basics you need to start off. From here you can proceed to find out more about your computer by reading books which tell you how the Electron works, as well as by carrying out experiments.

Alternatively you can ignore how the computer works and learn how to make it do exciting things. If you decide to follow this path you need to look at books on different programming languages. In this way you see what toolkits are available for building your game. So far you have been introduced to the BASIC language, but there are plenty more such as Pascal, FORTH, LOGO, PROLOG and **assembly language**. This last one, like BASIC, comes free with the machine; it is very close to the computer's internal 'machine code'.

Each programming language is designed with some particular kinds of application in mind. As a result, learning a new language introduces you to a world of new things which it would be hard to do (or think of doing) in BASIC.

Our group at Exeter University is developing a series of books to introduce novices to programming in different languages. Similar books to this one, but for other languages, should soon be available; and Acorn are producing the system which the Electron would need to run most of the languages we have mentioned.

BASIC, however, can be useful for a little while longer, and we are currently developing a follow-up book which is concerned with other things you can do with your computer. Here we have just touched on the world of games; there we shall develop this theme. We also want to see if there is any relationship between 'computing' and 'thinking'! We shall try to make the computer play the 'thinking game'.

In the meantime let your computer ponder on the answer to the question of 'The meaning of Life, the Universe and Everything.' Write a procedure which will respond suitably to:

```
>PROCQUESTION("The meaning of Life, the Universe  
and Everything.")
```

Appendix A

Turtle Graphics

```
10 REM M-TEXTURT
20 REM Optimised for space.
30 REM Stephen Goudge, October 1982
40
50 DIM direction%(7,2),dirname$(7),world% 600
60 MODE 4
70 PROCinitialise
80 PROCsetup
90 END
100
110
120 DEFPROCinitialise
130 LOCAL cns%,cnt%
140 REM Character definitions
150 VDU 23,224,16,56,84,146,146,16,16,16
160 VDU 23,225,16,16,16,146,146,84,56,16
170 VDU 23,226,24,32,64,255,64,32,24,0
180 VDU 23,227,24,4,2,255,2,4,24,0
190 VDU 23,228,252,192,160,144,136,132,2,1
200 VDU 23,229,128,64,33,17,9,5,3,63
210 VDU 23,230,63,3,5,9,17,33,64,128
220 VDU 23,231,1,2,132,136,144,160,192,252
230 VDU 23,232,24,60,126,255,24,24,24,24
240 VDU 23,233,24,24,24,24,255,126,60,24
250 VDU 23,234,16,48,112,255,255,112,48,16
260 VDU 23,235,8,12,14,255,255,14,12,8
270 VDU 23,236,124,248,240,248,204,134,3,1
280 VDU 23,237,128,192,97,51,31,15,31,62
290 VDU 23,238,62,31,15,31,51,97,192,128
300 VDU 23,239,1,3,134,204,248,240,248,124
310 VDU 23,240,255,255,255,255,255,255,255
330 xoffset%=0:yoffset%=21
340 maxx%=30:maxy%=20
350 xmax%=33:xmin%=2:ymax%=24:ymin%=3
360 FOR cnt%=0 TO 7
370 READ direction%(cnt%,0),direction%(cnt%,1),directi
on%(cnt%,2)
```



```

380   READ dirname$(cnt%)
390   NEXT cnt%
400 ENDPROC
410
420 DATA 23,224,232,"North      "
430 DATA 33,230,238,"North East"
440 DATA 32,227,235,"East      "
450 DATA 31,229,237,"South East"
460 DATA 21,225,233,"South      "
470 DATA 11,231,239,"South West"
480 DATA 12,226,234,"West       "
490 DATA 13,228,236,"North West".
500
510 DEFPROCsetup
520 LOCAL cnt%
530 MOVE 40,948
540 DRAW 1104,948:DRAW 1104,200
550 DRAW 40,200:DRAW 40,948
560 MOVE 88,904
570 DRAW 1056,904:DRAW 1056,248
580 DRAW 88,248:DRAW 88,904
590 DRAW 40,948
600 MOVE 1056,904:DRAW 1104,948
610 MOVE 1056,248:DRAW 1104,200
620 MOVE 88,248:DRAW 40,200
630 VDU 5
640 REM Draw in numeric scale.
650 MOVE 96,940
660 FOR cnt%=1 TO 30:PRINT;cnt% MOD 10;:NEXT cnt%
670 MOVE 96,240
680 FOR cnt%=1 TO 30:PRINT;cnt% MOD 10;:NEXT cnt%
690 MOVE 48,280
700 FOR cnt%=1 TO 20
710   PRINT;cnt% MOD 10;CHR$(8);CHR$(11);
720   NEXT cnt%
730 MOVE 1064,280
740 FOR cnt%=1 TO 20
750   PRINT;cnt% MOD 10;CHR$(8);CHR$(11);
760   NEXT cnt%
770 VDU 4
780 PRINTTAB(0,26);"Dir:           Pos:           Paint:' '
";
790 PROCLEARSCREEN
800 ENDPROC
810

```

```

820 DEFPROCLEARSCREEN
830 LOCAL cnt%
840 PROCNOERROR
850 dir%=0:xpos%=1:ypos%=1
860 xstart%=-1:ystart%=-1:xfinal%=-1:yfinal%=-1
870 pen%=2:paint$=""
880 FOR cnt%=0 TO 600
890     world%?cnt%=32
900     NEXT cnt%
910 xtext%=POS:ytext%=VPOS
920 VDU 28,xmin%+1,ymax%-1,xmax%-1,ymin%+1
930 CLS
940 PROCwritepaint
950 PROCwriteturtle
960 PROCwritedir
970 PROCwritepos
980 CLS
990 ENDPROC
1000
1010 DEFPROCtext
1020 VDU 28,0,30,39,27
1030 PRINTTAB(xtext%,ytext%);
1040 ENDPROC
1050
1060 DEFPROCdata
1070 xtext%=POS
1080 ytext%=VPOS
1090 VDU 28,0,26,39,26
1100 ENDPROC
1110
1120 DEFPROCplay
1130 xtext%=POS
1140 ytext%=VPOS
1150 VDU 28,xmin%,ymax%,xmax%,ymin%
1160 ENDPROC
1170
1180 DEFPROCwritedir
1190 PROCdata
1200 PRINTTAB(4,0);dirname$(dir%);
1210 PROCtext
1220 ENDPROC
1230
1240 DEFPROCwritepos
1250 PROCdata
1260 PRINTTAB(19,0);"      ";

```

```

1270 PRINTTAB(19,0); "(";xpos%;",",ypos%;")";
1280 PROCtext
1290 ENDPROC
1300
1310 DEFPROCwriteturtle
1320 PROCplay
1330 PRINTTAB(xoffset%+xpos%,yoffset%-ypos%);CHR$(direction%
on%(dir%,pen%));
1340 PROCtext
1350 ENDPROC
1360
1370 DEFPROCwritepaint
1380 PROCdata
1390 PRINTTAB(34,0);paint$;
1400 PROCtext
1410 ENDPROC
1420
1430 DEFPROCmove
1440 LOCAL ny%,nx%
1450 nx%=xpos%+INT(direction%(dir%,0)/10)-2
1460 ny%=ypos%+direction%(dir%,0) MOD 10-2
1470 IF (nx%<1)OR (ny%<1)OR (nx%>maxx%)OR (ny%>maxy%) THEN PR
OCError(9):ENDPROC
1480 IF pen%=2 THEN world%?((xpos%-1)*20+ypos%)=ASC(paint
$)
1490 PROCplay
1500 PRINTTAB(xoffset%+xpos%,yoffset%-ypos%);CHR$(world%?
((xpos%-1)*20+ypos%))
1510 xpos%=nx%;ypos%=ny%
1520 PROCtext
1530 PROCwriteturtle
1540 PROCwritepos
1550 ENDPROC
1560
1570 DEFPROCJUMPTO(x%,y%)
1580 IF error% THEN ENDPROC
1590 IF (x%<1)OR (y%<1)OR (x%>maxx%)OR (y%>maxy%) THEN PROCerr
or(10):ENDPROC
1600 PROCplay
1610 PRINTTAB(xoffset%+xpos%,yoffset%-ypos%);CHR$(world%?
((xpos%-1)*20+ypos%))
1620 xpos%=x%;ypos%=y%
1630 PROCtext
1640 PROCwriteturtle
1650 PROCwritepos

```

```

1660 ENDPROC
1670
1680 DEFPROCDRAWTO(x1%,y1%)
1690 LOCAL x,y,dx,dy,xt%,yt%,length%
1700 IF error% THEN ENDPROC
1710 length%=ABS(x1%-xpos%)
1720 IF ABS(y1%-ypos%)>length% THEN length%=ABS(y1%-ypos%
)
1730 dx=(x1%-xpos%)/length%;dy=(y1%-ypos%)/length%
1740 x=xpos%+0.5:y=ypos%+0.5
1750 IF length%=0 THEN ENDPROC
1760 REPEAT
1770     length%=length%-1
1780     xt%=INT(x):yt%=INT(y)
1790     IF (xt%>maxx%)OR(xt%<1)OR(yt%>maxy%)OR(yt%<1) THEN
PROCError(1):ENDPROC
1800     IF pen%=2 THEN world%?((xpos%-1)*20+ypos%)=ASC(pai
nt$)
1810     PROCplay
1820     PRINTTAB(xoffset%+xpos%,yoffset%-ypos%);CHR$(world
%?((xpos%-1)*20+ypos%));
1830     x=x+dx:y=y+dy
1840     xpos%=xt%:ypos%=yt%
1850     PROCtext:PROCwriteturtle:PROCwritepos
1860     UNTIL error% OR (length%<0)
1870 IF pen%=2 THEN world%?((xpos%-1)*20+ypos%)=ASC(paint
$)
1880 ENDPROC
1890
1900 DEFPROCFORWARD(num%)
1910 IF error% THEN ENDPROC
1920 IF num%<1 THEN PROCError(2):ENDPROC
1930 REPEAT
1940     num%=num%-1:PROCmove
1950     UNTIL error% OR (num%<1)
1960 ENDPROC
1970
1980 DEFPROCturnleft
1990 IF error% THEN ENDPROC
2000 dir%=(dir%+7) MOD 8
2010 PROCwriteturtle:PROCwritedir
2020 ENDPROC
2030
2040 DEFPROCturnright
2050 IF error% THEN ENDPROC

```

```

2060 dir%=(dir%+17) MOD 8
2070 PROCwriteturtle:PROCwritedir
2080 ENDPROC
2090
2100 DEFPROCLEFT(degrees%)
2110 LOCAL deg%
2120 IF error% THEN ENDPROC
2130 deg%=degrees% DIV 45
2140 IF (degrees%<>deg%*45)OR(degrees%<=0) THEN PROCError
(3):ENDPROC
2150 REPEAT
2160   PROCTurnleft
2170   deg%=deg%-1
2180   UNTIL (deg%=0)OR error%
2190 ENDPROC
2200
2210 DEFPROCRIGHT(degrees%)
2220 LOCAL deg%
2230 IF error% THEN ENDPROC
2240 deg%=degrees% DIV 45
2250 IF (degrees%<>deg%*45)OR(degrees%<=0) THEN PROCError
(4):ENDPROC
2260 REPEAT
2270   PROCTurnright
2280   deg%=deg%-1
2290   UNTIL (deg%=0)OR error%
2300 ENDPROC
2310
2320 DEFPROCPENUP
2330 IF error% THEN ENDPROC
2340 pen%=1:PROCwriteturtle
2350 ENDPROC
2360
2370 DEFPROCPENDOWN
2380 IF error% THEN ENDPROC
2390 pen%=2:PROCwriteturtle
2400 ENDPROC
2410
2420 DEFPROCCHANGEPAINT(newp$)
2430 IF error% THEN ENDPROC
2440 IF LEN(newp$)>1 THEN PROCError(5):ENDPROC
2450 IF ASC(newp$)<32 THEN PROCError(6):ENDPROC
2460 paint$=newp$
2470 PROCwritepaint
2480 ENDPROC

```

```

2490
2500 DEFFNAHEAD
2510 LOCAL x%,y%
2520 x%=xpos%+INT(direction%(dir%,0)/10)-2
2530 y%=ypos%+direction%(dir%,0) MOD 10 -2
2540 IF (x%<1)OR(y%<1)OR(x%>maxx%)OR(y%>maxy%)THEN =CHR$(
0) ELSE =CHR$(world%?((x%-1)*20+y%))
2550
2560 DEFFNUNDER
2570 =CHR$(world%?((xpos%-1)*20+ypos%))
2580
2590 DEFFNXPOS=xpos%
2600
2610 DEFFNYPOS=ypos%
2620
2630 DEFPROCNOERROR
2640 error%= FALSE
2650 ENDPROC
2660
2670 DEFPROCerror(num%)
2680 PROCText
2690 error%=TRUE
2700 IF num%=1 THEN PRINT "I'VE HIT A MAZE WALL"
2710 IF num%=2 THEN PRINT "I CAN NOT GO FORWARD LIKE THAT
"
2720 IF num%=3 THEN PRINT "I CAN NOT TURN LEFT LIKE THAT"
"
2730 IF num%=4 THEN PRINT "I CAN NOT TURN RIGHT LIKE THAT
"
2740 IF num%=5 THEN PRINT "ONE CHARACTER ONLY FOR A PAINT
"
2750 IF num%=6 THEN PRINT "NON-PRINTING PAINTS NOT ALLOWE
D"
2760 IF num%=8 THEN PRINT "NO SUCH MAZE"
2770 IF num%=9 THEN PRINT "I'VE HIT THE EDGE OF THE WORLD
"
2780 IF num%=10 THEN PRINT "I CAN'T JUMP THERE"
2790 PROCNOERROR:END
2800 ENDPROC
2810
2820 REM Short-form names
2830 DEFPROCFD(X%):PROCFORWARD(X%):ENDPROC
2840 DEFPROCRT(X%):PROCRIGHT(X%):ENDPROC
2850 DEFPROCLT(X%):PROCLEFT(X%):ENDPROC
2860 DEFPROCPU:PROCPENUP:ENDPROC

```

```
2870 DEFPROCPD:PROCPENDOWN:ENDPROC
2880 DEFPROCCP(X$):PROCCHANGEPAINT(X$):ENDPROC
2890 DEFPROCCS:PROCCLEARSCREEN:ENDPROC
2900 DEFPROC DR(X%,Y%): PROCDRAWTO(X%,Y%): ENDPROC
2910 DEFPROCJP(X%,Y%):PROCJUMPTO(X%,Y%):ENDPROC
2920 DEFFNHEADING=dir%*45
2930 DEFFNHE=dir%*45
2940 DEFFNAH=FNAHEAD
2950 DEFFNUN=FNUNDER
2960 DEFFNXP=FNXPOS
2970 DEFFNYP=FNYPPOS
```

>

Appendix B

Maze Solver

IMPORTANT NOTE

This program uses the TURTLE GRAPHICS program up to and including line 2490. MAZE SOLVER then continues as shown in the following listing.

```
2500 DEFPROCstart(x%,y%)
2510 PROCJP(x%,y%)
2520 xstart%=x%:ystart%=y%
2530 ENDPROC
2540
2550 DEFFNXSTART=xstart%
2560 DEFFNYSTART=ystart%
2570 DEFPROCfin(x%,y%)
2580 world%?((x%-1)*20+y%)=ASC("F")
2590 xfin%=x%:yfin%=y%
2600 PROCplay
2610 PRINTTAB(xoffset%+x%,yoffset%-y%);"F";
2620 PROCText
2630 PROCCP(". ")
2640 ENDPROC
2650
2660 DEFFNXFINISH=xfin%
2670
2680 DEFFNYFINISH=yfin%
2690
2700 DEFPROCGETMAZE(m%)
2710 DEFPROCGM(m%)
2720 LOCAL first%,second%,halt%
2730 PROCCS
2740 PROCCP(CHR$(240))
2750 IF(m%<1) OR (m%>7) THEN PROCError(11):ENDPROC
2760 IF m%=1 THEN RESTORE 2970
2770 IF m%=2 THEN RESTORE 2980
2780 IF m%=3 THEN RESTORE 2990
2790 IF m%=4 THEN RESTORE 3030
```



```

2800 IF m%=5 THEN RESTORE 3060
2810 IF m%=6 THEN RESTORE 3120
2820 IF m%=7 THEN RESTORE 3180
2830 halt%=FALSE
2840 REPEAT
2850   READ first%,second%
2860   IF first%=-10 THEN halt%=TRUE:GOTO 2900
2870   PROCJP(first%,second%)
2880   READ first%,second%
2890   PROCDR(first%,second%)
2900   UNTIL halt%
2910 READ first%
2920 PROCstart(second%,first%)
2930 READ first%,second%
2940 PROCfin(first%,second%)
2950 ENDPROC
2960
2970 DATA 1,8,20,8,-10,4,2,6,14
2980 DATA 8,4,8,20,22,1,22,16,8,4,18,4,12,16,22,16,-10,4,
10,26,10
2990 DATA 16,2,26,2,14,4,24,4,2,6,16,6,4,8,14,8,8,10,14,1
0,10,12,16,12,10,19,24,19
3000 DATA 2,6,2,20,4,8,4,20,8,10,8,20,10,12,10,19,14,1,14
,4,14,8,14,10
3010 DATA 16,1,16,2,16,6,16,12,24,4,24,19,26,2,26,20
3020 DATA -10,15,1,3,20
3030 DATA 6,2,18,2,8,4,18,4,8,8,22,8,6,10,20,10,1,14,12,1
4,1,16,10,16
3040 DATA 12,19,20,19,6,2,6,10,8,4,8,8,10,16,10,20,12,14,
12,19
3050 DATA 18,2,18,4,20,10,20,19,22,8,22,20,-10,1,15,17,3
3060 DATA 2,2,12,2,16,2,26,2,2,4,24,4,2,6,16,6,18,6,22,6,
4,8,14,8
3070 DATA 8,10,14,10,16,10,22,10,10,12,24,12,12,14,24,14,
10,16,22,16
3080 DATA 10,18,24,18,26,18,28,18,10,20,30,20,2,2,2,4,2,6
,2,20
3090 DATA 4,8,4,20,8,10,8,20,10,16,10,18,14,1,14,4,14,8,1
4,10,16,1,16,2
3100 DATA 16,6,16,10,18,6,18,10,20,7,20,8,24,4,24,12,24,1
4,24,18,26,2,26,18
3110 DATA 28,2,28,18,30,1,30,20,-10,15,1,3,20
3120 DATA 1,20,8,20,2,2,12,2,16,2,26,2,2,4,24,4,2,6,16,6
3130 DATA 18,6,22,6,4,8,14,8,8,10,14,10,16,10,22,10,10,12
,24,12,10,16,23,16

```

```

3140 DATA 12,14,24,14,2,18,8,18,10,18,24,18,26,18,28,18,1
2,20,28,20
3150 DATA 2,2,2,4,2,6,2,18,4,8,4,18,8,10,8,18,10,12,10,19
,14,1,14,4,14,8
3160 DATA 14,10,16,1,16,2,16,6,16,10,18,6,18,10,24,4,24,1
2,26,2,26,18,28,2
3170 DATA 28,18,30,1,30,19,-10,15,1,3,17
3180 DATA 10,2,29,2,10,4,24,4,26,6,28,6,12,8,18,8,16,12,1
8,12,20,12,28,12
3190 DATA 6,14,8,14,16,14,28,14,10,16,22,16,4,19,14,19,16
,18,26,18
3200 DATA 2,1,2,10,2,12,2,19,4,1,4,19,8,2,8,10,10,6,10,16
,12,8,12,12,18,8
3210 DATA 18,12,24,4,24,10,28,6,28,18,-10,1,1,14,10
3220
3230 DEFFNAHEAD
3240 LOCAL x%,y%
3250 x%=xpos%+INT(direction%(dir%,0)/10)-2
3260 y%=ypos%+direction%(dir%,0) MOD 10 -2
3270 IF (x%<1)OR(y%<1)OR(x%>maxx%)OR(y%>maxy%)THEN =CHR$(
0) ELSE =CHR$(world%?((x%-1)*20+y%))
3280
3290 DEFFNUNDER
3300 =CHR$(world%?((xpos%-1)*20+ypos%))
3310
3320 DEFFNFINISHED
3330 =((xpos%=xfin%)AND(ypos%=yfin%))
3340
3350 DEFFNXPOS=xpos%
3360
3370 DEFFNYPOS=ypos%
3380
3390 DEFPROCNOERROR
3400 error%=FALSE
3410 ENDPROC
3420
3430 DEFPROCerror(num%)
3440 PROCText
3450 error%=TRUE
3460 IF num%=1 THEN PRINT "I'VE HIT A MAZE WALL"
3470 IF num%=2 THEN PRINT "I CAN NOT GO FORWARD LIKE THAT"
"
3480 IF num%=3 THEN PRINT "I CAN NOT TURN LEFT LIKE THAT"
"
3490 IF num%=4 THEN PRINT "I CAN NOT TURN RIGHT LIKE THAT"
"

```

```

3500 IF num%=5 THEN PRINT "ONE CHARACTER ONLY FOR A PAINT
"
3510 IF num%=6 THEN PRINT "NON-PRINTING PAINTS NOT ALLOWE
D"
3520 IF num%=8 THEN PRINT "NO SUCH MAZE"
3530 IF num%=9 THEN PRINT "I'VE HIT THE EDGE OF THE WORLD
"
3540 IF num%=10 THEN PRINT "I CAN'T JUMP THERE"
3550 PROCNOERROR:END
3560 ENDPROC
3570
3580 REM Short-form names
3590 DEFPROCFD(X%):PROCFORWARD(X%):ENDPROC
3600 DEFPROCRT(X%):PROCRIGHT(X%):ENDPROC
3610 DEFPROCLT(X%):PROCLEFT(X%):ENDPROC
3620 DEFPROCPU:PROCPENUP:ENDPROC
3630 DEFPROC PD:PROCPENDOWN:ENDPROC
3640 DEFPROCCP(X$):PROCCHANGEPAINT(X$):ENDPROC
3650 DEFPROCCS:PROCCLEARSCREEN:ENDPROC
3660 DEFPROC DR(X%,Y%):PROCDRAWTO(X%,Y%):ENDPROC
3670 DEFPROCJP(X%,Y%):PROCJUMPTO(X%,Y%):ENDPROC
3680 DEFPROCGM(X%):PROCGETMAZE(X%):ENDPROC
3690 DEFFNAH=FNAHEAD
3700 DEFFNUN=FNUNDER
3710 DEFFNHEADING=dir%*45
3720 DEFFNHE=dir%*45
3730 DEFFNXS=FNXSTART
3740 DEFFNXF=FNXFINISH
3750 DEFFNYS=FNystart
3760 DEFFNYF=FNyFINISH
3770 DEFFNXP=FNxPOS
3780 DEFFNYP=FNyPOS
3790 DEFFNFIN=FNFINISHED
3800
3810 REM The maze solving routines.
3820
3830 DEFFNwall
3840 =((FNAH=CHR$(0))OR(FNAH=CHR$(240)))
3850
3860 DEFFNleftwall
3870 LOCAL h%
3880 PROCLT(90)
3890 h%=FNwall
3900 PROCRT(90)
3910 =h%

```

```

3920
3930 DEFFNrightwall
3940 LOCAL h%
3950 PROCRT(90)
3960 h%=FNwall
3970 PROCLT(90)
3980 =h%
3990
4000 DEFFNdeadend
4010 LOCAL h%, i%
4020 h%=0
4030 PROCLT(90)
4040 FOR i%=1 TO 3
4050     IF FNwall THEN h%=h%+1
4060     PROCRT(90)
4070     NEXT
4080 PROCLT(90)
4090 =(h%=3)
4100
4110 DEFPROChugleft
4120 IF NOT FNleftwall THEN PROCLT(90):PROCFD(1):ENDPROC
4130 IF (FNwall AND NOT FNrightwall) THEN PROCRT(90):PROC
FD(1):ENDPROC
4140 IF NOT FNwall THEN PROCFD(1):ENDPROC
4150 PROCRT(180):PROCFD(1)
4160 ENDPROC
4170
4180 DEFPROCTRYHUGLEFT
4190 REPEAT
4200     PROChugleft
4210     UNTIL FNFIN
4220 ENDPROC
4230
4240 DEFPROChugright
4250 IF NOT FNrightwall THEN PROCRT(90):PROCFD(1):ENDPROC

4260 IF (FNwall AND NOT FNleftwall) THEN PROCLT(90):PROCF
D(1):ENDPROC
4270 IF NOT FNwall THEN PROCFD(1):ENDPROC
4280 PROCRT(180):PROCFD(1)
4290 ENDPROC
4300
4310 DEFPROCTRYHUGRIGHT
4320 REPEAT
4330     PROChugright

```

```
4340 UNTIL FNFIN
4350 ENDPROC
4360
4370 DEFPROCTRY(N%)
4380 PROCGM(N%)
4390 PROCCP("@")
4400 PRINT"Left wall hugging."
4410 PROCTRYHUGLEFT
4420 PROCJP(FNXS,FNYS)
4430 PROCCP("#")
4440 PRINT"Right wall hugging."
4450 PROCTRYHUGRIGHT
4460 ENDPROC
>
```

Appendix C

Greeter Program

```
10 DIM NAME$(20), RMARK$(20)
20 NUMBER_OF_PEOPLE=0
30 CLS
40 PROCGREET
50
60 DEF PROCGREET
70 FOUND=FALSE
80 PRINT"" "Good morning! How are you"
100 INPUT REPLY$
130 PROCHECK("ILL", "I'm sorry to hear that.")
140 PROCHECK("NOT", "I'm not feeling too good myself, a
ctually.")
150 PROCHECK("VERY WELL", "That is good to hear!")
160 IF FOUND=FALSE THEN PRINT "I am quite well too, than
k you."
170 PROCIDENTIFY
175 PROCGREET
180 ENDPROC
190
200 DEF PROCHECK(TEST$, RMARK$)
205 IF LEN(REPLY$)<LEN(TEST$) THEN ENDPROC
210 IF INSTR(REPLY$,TEST$)<>0 THEN PRINT RMARK$: FOUND=T
RUE
220 ENDPROC
230
240 DEF PROCIDENTIFY
260 RECOGNISED=FALSE
270 PRINT "What name are you usually called by"
280 INPUT REPLY$
290 PRINT "Ah! You are called "REPLY$". Hello!"
300 FOR LOOP = 1 TO NUMBER_OF_PEOPLE
310 IF REPLY$=NAME$(LOOP) THEN PRINT RMARK$(LOOP)"! I
remember!": RECOGNISED=TRUE
320 NEXT LOOP
330 IF RECOGNISED=FALSE THEN PROCNEWPERSON(REPLY$)
335 PRINT "May I greet the next person, please?"
340 ENDPROC
350
```

```
360 DEF PROCNEWPERSON(NAME$)
370 PRINT"I don't think I have met you before."
380 PRINT"What shall I say when I meet you next time?"
390 INPUT RMARK$
400 NUMBER_OF_PEOPLE=NUMBER_OF_PEOPLE+1
410 NAME$(NUMBER_OF_PEOPLE)=NAME$
420 RMARK$(NUMBER_OF_PEOPLE)=RMARK$
430 PRINT"Thank you! I shall remember that."
440 ENDPROC
```

>

Appendix D

Rivergame Program

```
10 REM RIVERGAME
20 REM WITHOUT GRAPHICS
30 REM Michael Coker, December 1982
40 MODE 6
50 DIM POSITION$(1,3),TEST$(7)
60 POSITION$(0,1)="LB":POSITION$(0,2)="B"
70 POSITION$(0,3)="RB":FIRST=TRUE
80 CLS
90 PRINT"You are a man with a boat and a "
100 PRINT"chicken, a fox and a bag of grain"
110 PRINT"which you must get to the other side"
120 PRINT"of a river, using your little boat."
130 PRINT
140 PRINT"Each object, including yourself,may be "
150 PRINT"      at the right bank (RB)"
160 PRINT"      at the left bank (LB)"
170 PRINT"      or in the boat (BOAT)"
180 PRINT"and you can move them around as you "
190 PRINT"wish. You must remember that the FOX"
200 PRINT"will eat the CHICKEN if you are"
210 PRINT"not there; and the CHICKEN will"
220 PRINT"eat the GRAIN if left alone with it."
230 PRINT
240 PROCSTARTOFF
250 PROCWHATNOW
260 END
270 REM      ///// End of initialisation
280 DEF PROCSTARTOFF
290 PRINTCHR$(130)"PROCSTARTOFF"
300 POSITION$(1,1)="++C+M+F+G+B+"
310 POSITION$(1,2)="(++)"
320 POSITION$(1,3)="++"
330 BOAT$="(++)= "
340 PRINT"Press any key to go on."
350 IF GET$="" THEN 350
360 CLS
370 ENDPROC
380 REM      ///// Above: shows the start
```



```

390 DEF PROCSHOWIT
400 PRINTCHR$(130)"PROCSHOWIT"
410 PRINT''''''''
420 PRINT"(1)LEFTBANK"TAB(14)"(2)RIVER" TAB(28)"(3)RIGHTBA
NK''''
430 IF FIRST=TRUE THEN PRINTTAB(14)"boat"
440 PRINT POSITION$(1,1); "½";BOAT$; "½";POSITION$(1,3)
450 PRINTSTRING$(38,CHR$(47))
460 FIRST=FALSE
470 ENDPROC
480 REM      /////
490 REM      /////
500 DEF PROCWHATNOW
510 PRINTCHR$(130)"PROCWHATNOW"
520 IF LEN(POSITION$(1,3))>11 THEN PRINT"YOU HAVE DONE IT!
CONGRATULATIONS!":END
530 PROCSHOWIT
540 INPUT"WHAT SHALL I MOVE?(G,F,C,M,B)      " THING$

550 IF THING$="" THEN540
560 INPUT"WHERE TO?(LB,B,RB)      " PLACE$
570 IF PLACE$="" THEN560
580 PROCMOVE(THING$,PLACE$)
590 PROCWHATNOW
600 ENDPROC
610 REM      ///// Above: the user interface
620 DEF PROCMOVE(WHAT$,WHERE$)
630 PRINTCHR$(130)"PROCMOVE"
640 PROCILLEGALMOVE(WHAT$,WHERE$)
650 FOR LOOP=1 TO 3
660   IF INSTR(POSITION$(1,LOOP),WHAT$)<>0 THEN POSITION$(
1,LOOP)=FN TAKEOUT(WHAT$,POSITION$(1,LOOP))
670   IF WHERE$=POSITION$(0,LOOP) AND INSTR(POSITION$(1,LO
OP),WHAT$)=0 THEN POSITION$(1,LOOP)=FNPUTIN(WHAT$,POSITION$(
1,LOOP))
680   NEXT LOOP
690 PROCDISASTROUSMOVE(WHAT$,WHERE$)
700 PROCBOATPOS
710 ENDPROC
720 REM      /////
730 DEF PROCILLEGALMOVE(WHAT$,WHERE$)
740 PRINTCHR$(130)"PROCILLEGALMOVE"
750 LOOP=0: WRONGPLACE=TRUE: WRONGTHING=TRUE: WRONGSIDE=FA
LSE
760 HELPLESSBOAT=FALSE: MANABSENT=FALSE: BOATLESS=FALSE

```

```

770 FOR LOOP=1 TO 3
780 IF WHAT$<>"B" AND WHERE$<>"B" AND POSITION$(0,LOOP)=
WHERE$ AND INSTR(POSITION$(1,LOOP),"B")=0 THEN BOATLESS=TRUE
790 IF WHAT$<>"B" AND INSTR(POSITION$(1,LOOP),WHERE$)<>0
AND INSTR(POSITION$(1,2),"M")<>0 THEN MANABSENT=TRUE
800 IF WHERE$=POSITION$(0,LOOP) AND INSTR(POSITION$(1,LO
OP),WHAT$) THEN PROCALREADYTHERE(WHAT$,WHERE$): PROCWHATNOW
810 IF WHERE$=POSITION$(0,LOOP) THEN WRONGPLACE=FALSE
820 IF INSTR(POSITION$(1,LOOP),WHAT$)<>0 THEN WRONGTHING=
FALSE
830 IF WHERE$="B" AND INSTR(POSITION$(1,LOOP),WHAT$)=0 A
ND INSTR(POSITION$(1,LOOP),"B")<>0 THEN WRONGSIDE=TRUE
840 NEXT LOOP
850 IF WHAT$="B" AND INSTR(POSITION$(1,2),"M")=0 THEN HELP
LESSBOAT=TRUE
860 IF WRONGPLACE=TRUE THEN PRINTCHR$(129)"I DON'T RECOGNI
SE "WHERE$: PROCWHATNOW
870 IF BOATLESS=TRUE THEN PRINTCHR$(129)"ILLEGAL ATTEMPT T
O CROSS WITHOUT BOAT!": PROCWHATNOW
880 IF WRONGTHING=TRUE THEN PRINTCHR$(129)"I DON'T KNOW WH
AT A ";WHAT$;" IS.": PROCWHATNOW
890 IF HELPLESSBOAT=TRUE THEN PRINTCHR$(129)"BOAT NEEDS SO
MEBODY TO ROW HER!": PROCWHATNOW
900 IF WRONGSIDE=TRUE THEN PRINTCHR$(129)"BOAT IS AT THE W
RONG SIDE. CANNOT DO IT!":PROCWHATNOW
910 IF MANABSENT=TRUE THEN PRINTCHR$(129)"MAN IS NOT HERE!
CANNOT DO THE MOVING!":PROCWHATNOW
920 ENDPROC
930 REM      /////
940 DEF PROCALREADYTHERE(WHAT$,WHERE$)
950 PRINTCHR$(130)"PROCALREADYTHERE"
960 LOCAL T1$,T2$
970 T1$=FNTRANSLATE(WHAT$)
980 T2$=FNTRANSLATE(WHERE$)
990 PRINT CHR$(129);T1$" IS ALREADY AT "T2$
1000 ENDPROC
1010 REM      /////
1020 DEF FNTRANSLATE(WORD$)
1030 REM PRINTCHR$(130)"FNTRANSLATE"
1040 RESULT$="":COUNTER=0
1050 RESTORE: REPEAT
1060 COUNTER=COUNTER+1
1070 READ TEST$(COUNTER)
1080 IF LEFT$(TEST$(COUNTER),1)=LEFT$(WORD$,1) THEN RESUL
T$=TEST$(COUNTER)

```

```

1090 UNTIL COUNTER=7 OR RESULT$<>"
1100 DATA MAN, CHICKEN, FOX, GRAIN, BOAT, LBANK, RBANK
1110 =RESULT$
1120 REM      /////
1130 DEF FNTAKEOUT(THING$, CONTENT$)
1140 PRINTCHR$(130)"FNTAKEOUT"
1150 PLACE=INSTR(CONTENT$, THING$)
1160 LCHUNK$=LEFT$(CONTENT$, PLACE-1)
1170 RCHUNK$=RIGHT$(CONTENT$, LEN(CONTENT$)-PLACE-1)
1180 NEXCONTENT$=LCHUNK$+RCHUNK$
1190 REM IF LEN(NEXCONTENT$)<10 THEN NEXCONTENT$=STRING$(10
-LEN(NEXCONTENT$), " ")+NEXCONTENT$: REM*PROTECTION AGAINST B
UG IN <INSTR>
1200 =NEXCONTENT$
1210 REM      /////
1220 DEF FNPUTIN(THING$, PLACE$)
1230 PRINTCHR$(130)"FNPUTIN"
1240 RESULT$=LEFT$(PLACE$, INSTR(PLACE$, "+"))+THING$+"+"+RIG
HT$(PLACE$, LEN(PLACE$)-INSTR(PLACE$, "+"))
1250 =RESULT$
1260 REM      /////
1270 DEF PROCBOATPOS
1280 PRINTCHR$(130)"PROCBOATPOS"
1290 IF INSTR(POSITION$(1,1), "B")<>0 THEN BOAT$=POSITION$(1
,2)+STRING$(16-LEN(POSITION$(1,2)), "=")ELSE IF INSTR(POSITIO
N$(1,3), "B")<>0 THEN BOAT$=STRING$(16-LEN(POSITION$(1,2)), "="
)+POSITION$(1,2)ELSE PRINT"WHERE'S THE BOAT GONE, THEN?"
1300 ENDPROC
1310 REM      /////
1320 REM      /////
1330 DEF PROCDISASTROUSMOVE(WHAT$, WHERE$)
1340 PRINTCHR$(130)"PROCDISASTROUSMOVE"
1350 IF LEN(POSITION$(1,2))>8 THEN BOAT$="=====
":PRINTCHR$(129)"THE BOAT HAS SUNK I'M AFRAID. ONLY TWO":PRI
NT"PEOPLE ALLOWED!":PROCSHOWIT:PROCSTARTOFF
1360 IF FNSUPPER("F", "C")=FALSE AND FNSUPPER("C", "G")=FALSE
THEN ENDPROC
1370 ENDPROC
1380 REM      /////
1390 DEF FNSUPPER(ACTIVE$, PASSIVE$)
1400 PRINTCHR$(130)"FNSUPPER"
1410 EATEN=FALSE
1420 FOR LOOP=1 TO 3
1430 IF INSTR(POSITION$(1, LOOP), ACTIVE$)<>0 AND INSTR(POS
ITION$(1, LOOP), PASSIVE$)<>0 AND INSTR(POSITION$(1, LOOP), "M")

```

```
=0 THENPOSITION$(1,LOOP)=FNTAKEOUT(PASSIVE$,POSITION$(1,LOOP
)): EATEN=TRUE
1440     NEXT LOOP
1450 IF EATEN=TRUE THEN PRINTCHR$(129)"THE "FNTRANSLATE(PAS
SIVE$)" HAS BEEN EATEN BY THE "FNTRANSLATE(ACTIVE$): PROCBOA
TPOS: PROCSHOWIT: PROCSTARTOFF
1460 =EATEN
```

Index

Page numbers indicate where each new technical term is introduced.

- " 4
- \$ 67
- ↓ 6
- "2 4
- 6
- ← 6
- ?/ 4
- BREAK 3
- CAPS LK 3
- COPY 6
- DELETE 4
- ESCAPE 12
- FUNC 4
- G 21
- R 21
- RETURN 3
- SHIFT 3
- ↑ 6
- * 32
- *FX 90
- + 33
- 32
- / 33
- < 34
- <> 34
- = (function result) 36
- = (logical equivalence) 38
- = (with LET) 18
- > 34
- ^ 37
- Acorn 3
- AND 34
- argument 63
- array 69
- ASC 65
- ASCII codes 65
- assembly language 117
- assignment statement 33
- auto-repeat 88
- BASIC 3
- block number 20
- blocks 20
- bugs 7
- CHAIN 82
- CHR\$ 65
- CLG 96
- CLS 5
- compiler 15
- conditional branching 20
- coordinates (relative-) 103
- cursor (edit-) 6
- cursor (graphic-) 97
- DATA 58
- data 58
- debugging 7
- DEF 5
- DIM 69
- dimension 69
- DRAW 97
- editing 4
- editor 15
- Electron 3
- ELSE 19
- END 8
- ENDPROC 5
- ENVELOPE 86
- error messages 7
- error-trapping 90
- Exeter University 117
- expression 32
- FALSE 34
- FOR 18
- FORTH 117
- functions (inbuilt-) 35
- functions (own-) 35
- GCOL 105
- GOTO 90
- GREETER 81
- IF 19
- immediate mode 4
- information 57
- INKEY 89
- INPUT 60
- INSTR 77
- interpreter 15
- Introductory Cassette 24
- keyboard 1

keyword (BASIC-) 4	READ 58
LET 18	RECORD 20
LEFT\$ 77	recursion 12
LEN 74	REM 8
LIST 5	REPEAT 34
LOAD 20	reports 7
LOCAL 36	resolutions 95
LOGO 30	RESTORE 59
loop 18	RIGHT\$ 83
machine code 14	RND 35
mishap messages 7	RUN 9
MODE 93	SAVE 20
modes 93	SOLVER 46
MOVE 97	SOUND 84
name 18	SPC 63
NEW 12	SQR 35
NEXT 18	STEP 18
NOT 35	step number 87
null character 48	strings 66
OLD 37	structured programming 39
ON ERROR 90	subscript 69
Operating System 15	TAB 62
OR 34	THEN 19
Papert, S. 30	TO 18
pitch 84	top-down refinement 39
pitch increment 87	TRUE 34
pixel 96	TURTLE 24
PLAY 20	Turtle Graphics 30
PLOT 98	TV 3
PRINT 4	UNTIL 34
PRINTTAB 22	User Guide 3
PROC 5	value 18
procedure 5	variable 18
program (computer-) 1	VDU 95
PROLOG 117	voices 84
prompt 3	

START PROGRAMMING WITH THE ELECTRON

If you've bought one of the new Acorn Electrons and you're a stranger to computing, then this book is the friend you need to hold your hand through a bewildering new world.

Start programming with the Electron provides a gentle introduction to what your new micro can do. It takes a structured approach to programming techniques, and new terms are explained as they are introduced. There's also a handy glossary of terms at the end of each chapter.

Once you've mastered the basics with this book, you'll be in a better position to make good use of the mass of useful information in the *User Guide*.

Features include:

- ★ problem solving
- ★ sound
- ★ graphics
- ★ games
- ★ using a 'turtle'
- ★ arithmetic on the Electron

At the back are complete listings of the programs used in the book.

Have fun!



Addison-Wesley Publishing Company

ISBN: 201 14604 5

